AFRL-RI-RS-TR-2011-218

PHOENIX: SERVICE ORIENTED ARCHITECTURE FOR INFORMATION MANAGEMENT - BASE IMPLEMENTATION DOCUMENT

SEPTEMBER 2011

INTERIM TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2011-218 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
STEVEN D. FARR
Branch Chief

/s/
JULIE BRICHACEK, Chief
Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
SEP 2011	Interim Technical Report	JAN 2009 – NOV 2010

4. TITLE AND SUBTITLE

PHOENIX: SERVICE ORIENTED ARCHITECTURE FOR INFORMATION MANAGEMENT - BASE IMPLEMENTATION DOCUMENT

5a. CONTRACT NUMBER
In House

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

V. Combs, J. Hanna, J. Bryant, B. Lipa, S. Tucker, T. Krokowski, J. Reilly, G. Hasseler

5d. PROJECT NUMBER
S2TS

5e. TASK NUMBER
IH

5f. WORK UNIT NUMBER
03

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFRL/RISE, 525 Brooks Road, Rome, NY 13441-4505
ITT, 775 Daedalian Drive, Rome NY 13440
RRC, Ridge Street, Rome NY 13440
ATC-NY, Thornwood Drive, Ithaca NY

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/Information Directorate
Rome Research Site
26 Electronic Parkway
Rome NY 13441

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2011-218

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2011-0021

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This document outlines the specifics of the Phoenix Base Implementation including technologies utilized and key design decisions. The Base Implementation consists of three segments: the component packages, the service packages, and the support packages. The component and service packages map functionally and semantically to their respective Phoenix Architecture packages of the same name. The support package contains the implementation specific packages designed to make utilization of the developed Phoenix services more convenient for the outside developer.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			VAUGHN COMBS
U	U	U	UU	76	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

Design

This document outlines the specifics of the Phoenix Base Implementation including technologies utilized and key design decisions. The Base Implementation consists of three segments: the component packages, the service packages, and the support packages. The component and service packages map functionally and semantically to their respective Phoenix Architecture packages of the same name. The support package contains the implementation specific packages designed to make utilization of the developed Phoenix services more convenient for the outside developer.

Conventions

This document provides both a literal and conceptual design of the Phoenix architecture. The literal architecture is a technical specification defined using UML. The conceptual architecture is a less formal description using plain language and diagrams to provide design concepts and objectives.

Diagram Conventions

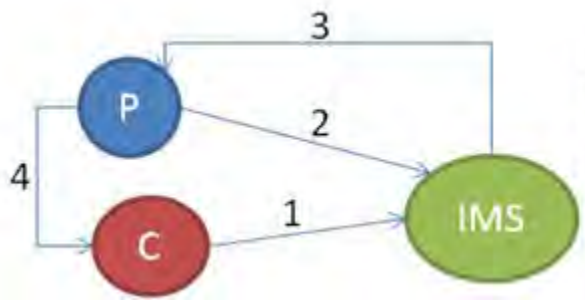
Throughout this document there are a number of non-UML diagrams that are used to illustrate high-level concepts. Samples of these diagrams are shown below along with usage information.

The figure below shows a sample communication between Phoenix entities via channels.

Entity	Meaning	Color
Producer	Produces information.	
Service	Manipulates information.	

Consumer	Consumes information.
Actor	A generic term that can mean producer, consumer, or service.
Inquisitor	A type of consumer that queries a service to get information.

The figure below is a sample diagram showing labeled information flow.



1. Consumer registers with IMS.
2. Producer requests consumer list.
3. Producer receives consumer list.
4. Producer delivers data directly to consumers.

Implementation Language

Java was selected as the development language of choice for this project due to several factors:

- Ease of Use and Understanding
- Existing Built-in Features including support for RMI, XML, and Concurrency
- Availability of numerous 3rd party libraries such as Log4J, XPP, among many others
- Development Team Experience

At design time the decision was made to go with the latest version of the Java Software Development Kit (SDK) available, which was Java Developer's Kit (JDK) 6. JDK 7 could not be considered because it is in an early development phase, which introduces too much risk and would inhibit engineering productivity.

Code Conventions and Formatting

Code formatting is a huge issue in a distributed development environment. Formatting has been standardized for the project by applying a standard format configuration file that is enforced through the activation of the Checkstyle plug-in for Eclipse and Maven. The current Eclipse plug-in version is 4.4.2 and the current Maven plug-in version is 2.2. More information about the Checkstyle Eclipse and Maven plug-ins can be found at the Checkstyle web-site: http://checkstyle.sourceforge.net

The current code format is an extension of Sun's suggested standard code conventions for Java applications. Some high level settings of note are a max line length of 120 characters, a max method size of 150 lines of code, and the application of variable declaration templates consistent with Sun's standards.

FindBugs : Bug Finding and Reporting Plug-in

Discovering bugs in a project is a job assigned to the FindBugs plug-in for Eclipse and Maven. Bugs are reported within Eclipse by the FindBugs views provided by the plug-in while the bugs reported by the Maven plug-in are available for view only through the Maven project module's individual web-sites. The Eclipse FindBugs plug-in can be configured to run during every compilation done by Eclipse while the Maven FindBugs plug-in is only run when Maven builds the module's corresponding web-site. The current versions of the FindBugs plug-in for Eclipse and Maven are 1.3.7 and 1.2, respectively. More information about the FindBugs Eclipse and Maven plug-ins can be found at the FindBugs web-site: http://findbugs.sourceforge.net

PMD

The Maven builder for the project also incorporates the PMD plug-in. This plug-in, run only when Maven builds the corresponding web-site for a project module, checks for possible bugs, dead and suboptimal code, overly complicated conditional expressions, and duplicate code. PMD reports are available via a link on each module's web-site. The current version of PMD Maven plug-in used for this project is 2.4. More information about the PMD plug-in and its capabilities can be found at its web-site: http://pmd.sourceforge.net

Component Package Implementations

An alphabetical listing of the component package implementations and their specifics:

- Channel
- Core
- Event
- Expression
- Information
- Service
- Session
- Stream

The component packages of the Base Implementation contain the low level entities that give life to the service implementations. These include the definitions of contexts (Base Context), services (Base Service and Base Channel Service), information (Information), events (Event and sub-classes), and actors (Session Context). Also included are the functional implementations of the byte, information, and event channels and expression processors.

The Base Implementation component packages have been broken into two categories: Definition and Functional. The Definition packages contain concrete classes that define Phoenix entities such as contexts, services, and information. Functional packages contain classes that implement basic functionalities (expression processing, data transport) that are utilized by the service packages to provide information management capabilities. Figure 1 shows a sliding scale of functional to definition packages with the mainly functional packages at the bottom of the scale and the primarily definition packages at the top. Most component packages fall within both

categories at the same time but not all. Those that do fit into both categories typically lean towards one end of the scale. Some packages (session and information) strictly adhere to the description of one category.

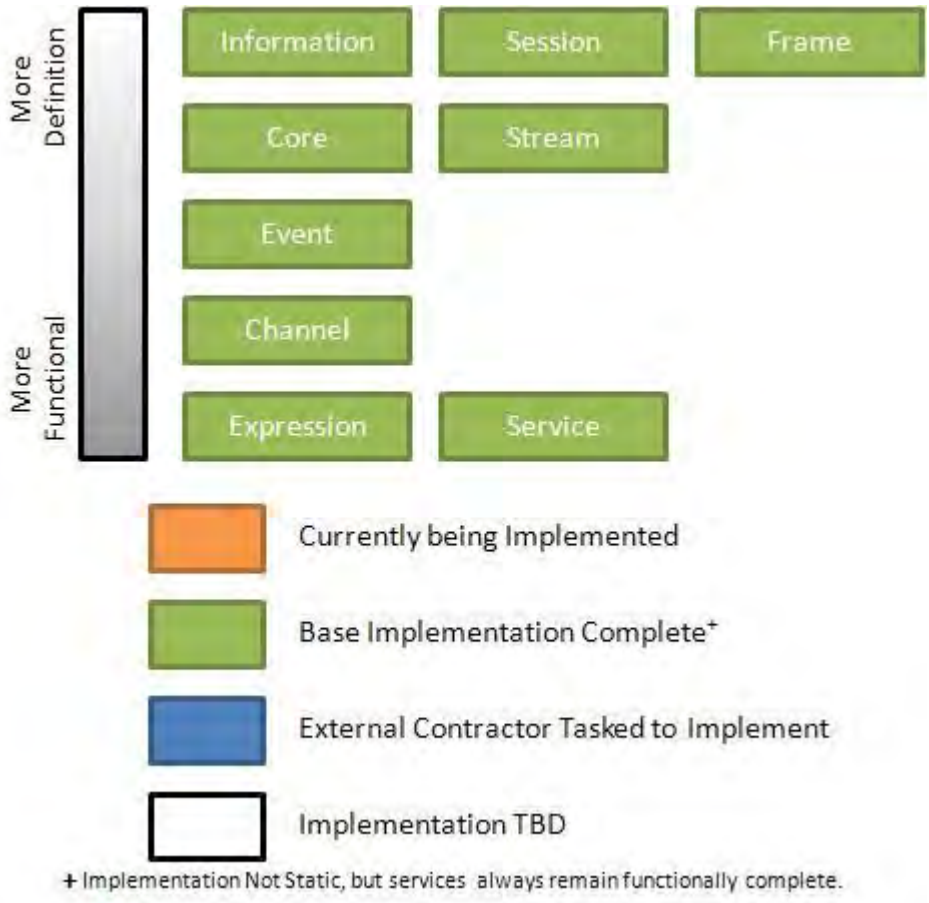


Figure 1 - Component Packages

The information and session packages are strictly definition packages because they only contain concrete entities that implement and define some of the key concepts within their respective Phoenix Abstract Architecture packages. The core package contains a plethora of definitions, such as the basic context implementation, but also falls into the functional category due to the basic implementation of the Base Service and Base Persistent Service interfaces. The event package contains the definition of the simple Event's outlined in the Phoenix Architecture, but also contains the event channel implementations used to move Event instances around. The channel implementation primarily contains the basic implementation of byte and information channels as well as the Base Channel Service, but it also contains the concrete implementations of the Channel, End Point, and Transport Contexts. The expression package defines the implementation of an Expression Context, but also provides the implementations of the Expression Processor interface used by the services.

Channel

The channel package provides the underlying mechanisms that enable data transfer among actors within the Phoenix architecture. It supports the core data types defined by the architecture; byte, event, frame, and information.

Input Channels

Common functionalities among input channels have been grouped into two levels. Functionalities common to all input channels have been located at the top level, in the AbstractInputChannel<T> class. At the next level down is a generic Phoenix input channel implementation, PhoenixInputChannel<T>. Finally, the last level in Figure 2 depicts the top level input channel constructs present in the architecture itself. These classes are composite classes, instantiating a private instance of PhoenixInputChannel<T>.

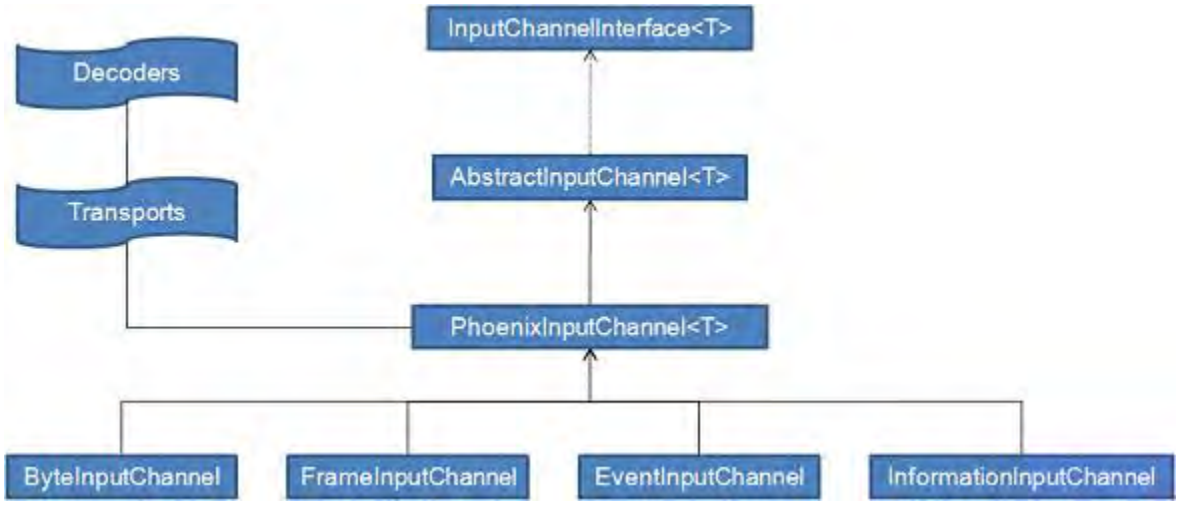


Figure 2 - Input Channel Inheritance

Output Channels

Like the input channels, common functionalities among output channels have also been grouped into two levels. Functionalities common to all output channels have been located at the top level in the AbstractOutputChannel<T> class. Again, at the next level down is a generic Phoenix output channel implementation, PhoenixOutputChannel<T>. Finally, the last level in Figure 3 contains the top level output channel constructs from the architecture itself. These output channel classes are also composite classes, containing a private instance of PhoenixOutputChannel<T>.

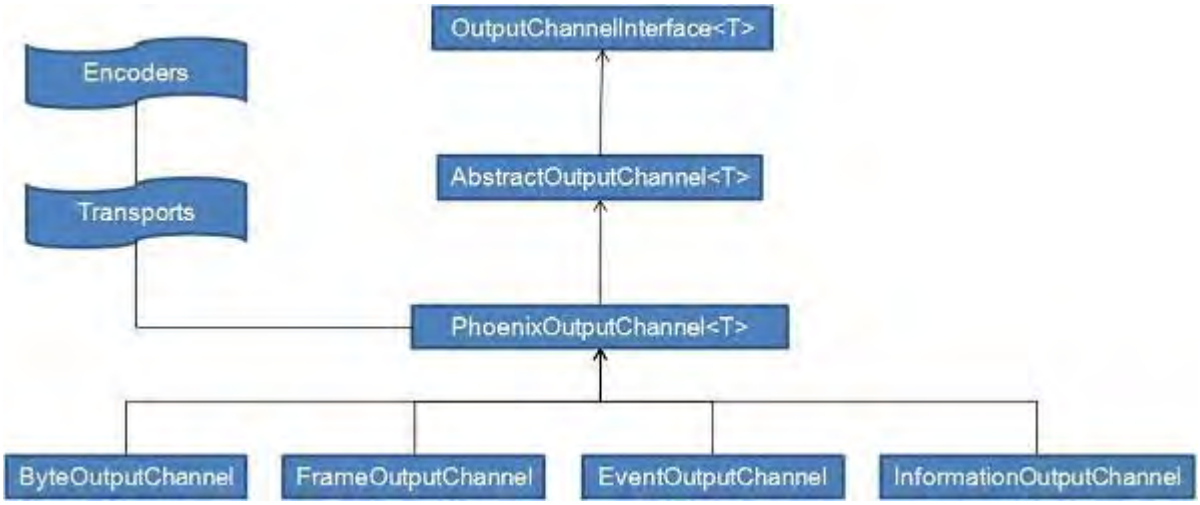


Figure 3 - Output Channel Inheritance

Channel Transports

In the transports implementation, two simple interfaces have been defined which allow for channel implementations to seamlessly select another transport at construction. The key interfaces here are ClientTransportInterface<T> and ServerTransportInterface<T>. Any new transport implementation may be added, so long as it implements these two interfaces. An example of a network-based transport that might be added at some point in the future is a Java sockets-based TCP transport.

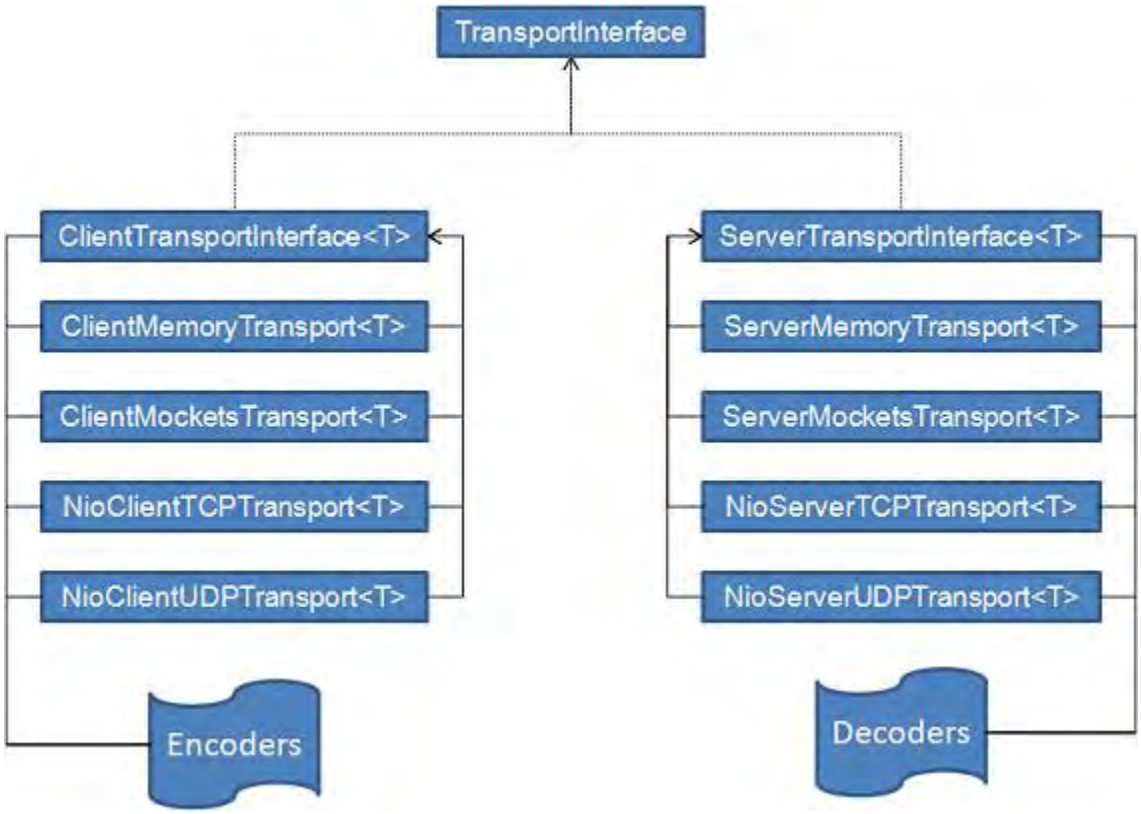


Figure 4 - Channel Transports Inheritance

Channel Handlers

The handler implementation has divided functionalities into two interfaces. The first interface, the HandlerInterface, is for handling exceptions. The second interface, InputHandlerInterface<T>, is for handling asynchronous reads. These interfaces are implemented in two classes, Handler and InputHandler<T>, respectively.

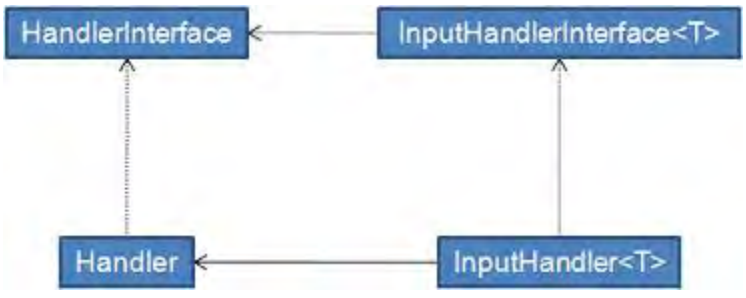


Figure 5 - Channel Handlers Inheritance

Channel Implementation

The channel implementation is composed of a number of sub-packages. The key sub-packages are contexts, handlers, and transports.

- **contexts** The contexts sub-package defines a number of contexts that are used within the channel package. These contexts include ones for configuring the high-level input and output channel abstraction, as well as supporting abstractions, such as end-points, application protocols, and transport protocols.
- **handlers** The handlers sub-package defines callback objects that are used by asynchronous channel operations.
- **transports** The transports sub-package provides a variety of low-level transports that are utilized by channels. It provides a memory-based transport (for communications within the same JVM), as well a variety of network-based transports and encoders and decoders.

Blocking and Non-Blocking IO

A single channel construct provides both blocking and non-blocking IO. Non-blocking IO methods are denoted by the suffix "Async," all other methods are assumed to be blocking.

Blocking methods, when called on a channel, block until their results are ready, and then immediately return their results. Non-blocking methods, however, return immediately after being called. The results of these method invocations are returned to an associated callback object, which this implementation refers to as handlers. Input channel based non-blocking methods return their results to a class implementing InputHandlerInterface<T>, to which they are provided a reference at the time of method invocation. Output channels, on the other hand, are associated with a class implementing HandlerInterface during construction, and non-blocking methods return their results to the associated handler.

Wire Format

All of the implemented transports package their payloads into instances of the MessagePacket<T> class. Typically, network-based transports will use MessagePacket<byte[]>, and memory-based transports will use MessagePacket<Object>. The MessagePacket class allows transports to communicate messages with different semantic meanings. This functionality becomes critical when performing such tasks as connection negotiation and connection management.

The supported message types are *WELCOME*, *WELCOME_RESPONSE*, *PAYLOAD*, and *GOODBYE*.

Encoding and Decoding

The message encoding and decoding interfaces dictate that the final encoding of an object be a byte[], and that the final decoding of an object originate from a byte[]. Encodings are set in the TransportContextInterface. Currently, only one encoding at a time is supported, despite the TransportContextInterface implying that multiple encodings may be selected.

The two encodings currently provided by base-implementation are *javaserial* and *xml*. *javaserial* is the default encoding.

Connection Negotiation

The process of connecting two channels is relatively straight-forward. An input channel must first be created from a ChannelContextInterface object. If a legal, non-zero port number has been specified, the channel will attempt to utilize this port; otherwise, a port will be selected by the system and the ChannelContextInterface object updated appropriately. In order for the newly created input channel to begin accepting connections, it must first be opened, which is done using the open method. The next step in the process is creating an output channel that will connect to the input channel. An output channel is created in the same way as an input channel, except the ChannelContextInterface object must contain a URI to a legal input channel, and its constructor must be passed a HandlerInterface object. The output channel may then be connected to the input channel by calling its connect method.

Upon receiving a new connection, the underlying server transport of the input connection sends a *WELCOME* packet/message containing a copy of its ChannelContextInterface object to the newly connected client transport of the connection output channel. Upon receiving the *WELCOME* message, the client sends an acknowledgement back to the server in the form of a *WELCOME_RESPONSE* message, containing a copy of its ChannelContextInterface object. From this point forward, the client sends *PAYLOAD* messages until it wishes to close the connection. When the client wishes to disconnect from the server, it sends a *GOODBYE* message to the server. This connection sequence is designed to allow an input channel to know which output channels are connected to it as well as allow for the possibility of performing dynamic connection reconfiguration.

Core

The core component package is the lowest common denominator of the Base Implementation code. This package includes the context, service, stub, and connector classes that form the base for all other component and service entities.

Contexts

Contexts are implemented as java.util.Map objects as this seemed the obvious choice for a Java object that utilizes name-value pairs. The Base Context class is implemented as an abstract class because it was determined at implementation design time that being able to instantiate a copy of a Base Context was not desired. Base Context instances would not be useful because all interfaces take specific sub-classes of contexts as parameters, not the generic Base Context.

Connectors and Stubs

RMI

The default implementation of Phoenix connectors and stubs was accomplished by using standard Java Remote Method Invocation (RMI). This was done because it is a technology that the design team is very familiar with and has used extensively. The core package implements the base level RMI connector and stub that all other service connectors and stubs extend, thereby inheriting the same exact RMI-specific connection code.

PIC

The Base Implementation also supports the ability to invoke services across language boundaries. The Phoenix Invocation Control (PIC) control channel allows information channel and control channel interactions across the Java and C++ language boundary. The PIC accomplishes this through a custom serialization of Phoenix objects.

Base Service

The implementation of the Base Service interface also implements the Base Persistent Service Interface so that there is one parent, abstract service class that provides both sets of functionalities. The persistence of service state is an important feature to share amongst all Phoenix service implementations. Implementing this interface within the Base Service class also allows for easier transition of the Base Channel Service interface to a cohesive implementation, at least within Java.

Base Attribute Update Callback

This callback is used by the implementation to notify registered entities of changes to specific attribute values witin a context. For example, if a Service Context has a set of callbacks registered with it that are triggered by the 'Service-State' attribute, whenever this attribute's value is changed the callbacks and their resident logic will be executed. It is envisioned that these callbacks could include policy driven logic decisions for other entities or messaging protocols to alert other entities of the change that occurred.

Event

The event component package contains the concrete constructs used to define what an event is. An event contains a body that is a generic Object. An event context also contains a field that is a generic Object, referred to as the event expression data, which is used by the ENS for brokering fired events over registered notification requests.

The event package also contains the Event specific channel implementations for moving Events around amongst Phoenix actors. The current set of supported transport level protocols is:

- memory - An implementation of channels that uses the memory space of the Java Virtual Machine (JVM) to transfer objects around.
- tcp - An implementation of the Transport Control Protocol (TCP) that uses the java.net.ServerSocket and java.net.Socket classes.
- udp - An implementation of the User Datagram Protocol (UDP) that uses the java.net.DatagramPacket and java.net.DatagramSocket classes.
- mocket - An implementation of the Mockets protocol defined by IHMC.

The Event Implementation Classes:

The current implementation of an event fully implements the Event interface as represented in the Phoenix architecture and base-interfaces. Event implementation classes may provide additional methods to retrieve all of parts of the message body in their native formats. In particular the class maintains the following attributes associated with an event:

- Event Context - This context describes additional detail about an event.
- Firing Actor ID - The session identifier of the actor firing this event.
- Event Body - The body of the event, any serializable object.

Event Notification Use Cases:

1. Consumer Hit Lists generated by Information Brokering,
2. Input Channel Status Updates from the Submission Service,
3. Output Channel Status Updates from the Dissemination Service,
4. Subscription Status Updates from the Information Brokering Service,
5. Submission of Information Acknowledgment and Negative Acknowledgment, and
6. Consumer Acknowledgement of Information Receipt

Current Set of Event Classes:

- **Consumer Hit List Event** - describes what subscriptions matched a brokered instance of information.
- **Information Acknowledgment Event** - acknowledge receipt of a specific information instance.
- **Information Watch Event** - signal an actor to watch for receipt of a specific information instance.
- **Input Channel Status Event** - status update for a specific input channel. Contains a (possibly modified) copy of the channel's context.
- **Output Channel Status Event** - status update for a specific output channel. Contains a (possibly modified) copy of the channel's context.
- **Submission Negative Acknowledgment Event** - alert event for signaling that a specific instance of information was not received within the amount of time specified by a previous Information Watch Event.
- **Subscription Status Event** - status update for a specific subscription. Contains a (possibly modified) copy of the subscription's context.

The current set of events also includes several sub-typing classes such as the Information, Information Type, and Exception event classes. These are higher level classes meant to be extended for specific operational use.

The event package also contains an EventChannelFactory implementation along with multiple EventInputChannel and EventOutputChannel implementations. The definitions and differences between block and stream implementations are may be found in the channel package documentation.

Expression

The expression implementation module contains the Base Implementation expression processors and contexts. All provided Base Implementation expression processors can be configured to either evaluate expressions and their related information types or just expressions (typed vs. untyped evaluation). All Base Implementation expression processors extend an Abstract Expression Processor class that defines common things such as the map containing the registered expressions. Due to the use of a map for maintaining the registry of expressions, evaluation order cannot be guaranteed for any Base Implementation processor. Base Implementation expression processors are as follows:

- **Context** - Supports brokering over context attributes. Assumes equals operations for all attributes unless the attribute is a Collection. Collections are assumed to be a contains operation. Assumes the AND conjunction for all operations, OR is currently not supported.
- **Expressionless** - Supports brokering over null expressions.
- **Regular Expression** - Supports all regular expressions via built in Java support.
- **XPath** - Supports all XPath 1.0 and most XPath 2.0 functions and capabilities via use of the XML Pull Parser version 3 (XPP3) library.

Expression Evaluation

Expressions are evaluated one at a time. If the expression matches, the associated expression context's name is added to the list of matching expression identifiers to be returned to the entity doing the evaluation. This name may be a subscription name (in the case of information brokering in the IBS it certainly is) or it may be some other identifier. The identifiers returned have no semantic meaning to the expression processors themselves but instead mean something to the entity doing the evaluation.

Frame

The frame package contains the components that provide the custom serialization, encapsulation, and stream sequencing (and dissemination) capabilities of the Base Implementation. These components include the frame object itself and the frame specific input and output channels.

Information

The information component package contains the embodiment of the central object of the Phoenix architecture, the information instance. This object contains four member variables and a bunch of set and retrieval methods. One of these variables is the Information Context. The information instance contains the metadata, payload, and information type identifier while the Information Context is used to store any additional descriptive data about the information instance such as the identifiers of the interested consumers and the degradation flag.

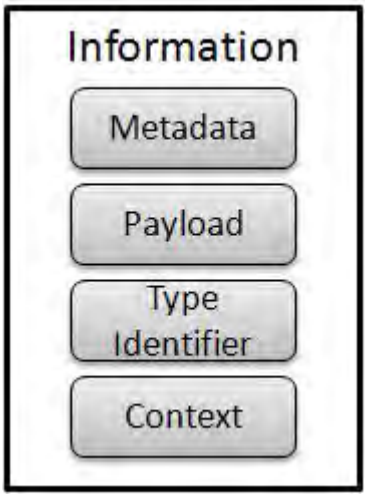


Figure 6 - Information Instance

Figure 6 shows the four elements that comprise an instance of information. Remember from the Phoenix Abstract Architecture documentation that an information instance can exist with any combination of these elements that contains at least one of the four fields shown above.

Metadata

Metadata within the Phoenix Base Implementation is an instance of a generic Java Object, requiring a specific casting operation to be performed to extract it in a native format (i.e. String, byte[]). Currently this is the main field utilized to describe the information instance by brokering and retrieval operations.

Payload

The payload of a Phoenix Base Implementation information instance has also been implemented using the generic Java Object. Future work may include the addition of payload processing to information brokering and retrieval operations.

Type Identifier

Phoenix Base Implementation information type identifiers are simple strings. This notion may be extended in the future to enable some form of hierarchical relationships among information type definitions, but none exist at the present. The Base Implementation does not understand or care if information type identifiers are single words or organized into packages such as Java classes typically are (i.e. 'mil.af.aircraft' and 'mil.n.ship' are treated as simple strings, nothing more). The empty string or a null value is used interchangeably by the Base Implementation to identify un-typed information. When performing information brokering, validation, persistence, and retrieval operations the information type identifier is utilized by the corresponding services in various ways. Information type identifiers in information instances should map to a single Information Type Context known to some Information Type Management Service. If not, the information will be treated as un-typed information and will be subject to the restrictions inherent with that tag when performing any or all of the aforementioned operations.

Context

The Base Implementation information instance's context has been extended beyond the architecture's definition of such a context by including a field for associated query identifier. This field is used to tag information instances being retrieved as part of a query's result set and has uses that include monitoring and tracking.

Service

The service package contains the components and utilities common to all Base Implementation services including the Channel Managers, Service Multiplexors, and Task Schedulers. All of these are pluggable for each unique service instance and are fully mappable to Spring-based configurations.

Control Channel Manager

The Control Channel Manager (CCM) provides a registry for service stubs. Each Phoenix service contains a single CCM that maintains the set of service stubs used by the CCM's parent service. For example, the CCM for a Submission Service (SS) may contain a set of Information Brokering Service (IBS) and Repository Service (RS) stubs. The CCM provides methods for adding, retrieving, and removing stubs as well as methods for connecting and disconnecting them. The

CCM methods for adding stubs provide the option of connecting the stub at the time of addition or at a later time.

Input & Output Managers

Input managers provide a boundary between the input channels and the actual service logic that processes them. The Base Implementation input managers are built upon the notion of a timer based buffer, which is a buffer that spends on a regular interval. The spend size, max buffer size, and spend interval are all configurable via a properties file (TimerBasedBuffer.properties). This file is not required, but if it is not found on the classpath all buffer's will utilize the hard-coded default settings. Buffer's are distinguishable by name. Output managers provide another break point between the service logic and the outgoing transmissions. Both input and output managers have been implemented through the use of the Java generics pattern, meaning that each instance of a manager may process a single, known object type. Input and output managers are maintained and controlled by a Phoenix service's Channel Manager.

Channel Manager

The Channel Manager (CM) is responsible for managing the parent service's input and output managers and channels. It also maintains the registry that maps channels to managers. Assigning an input channel to a manager causes any objects received by that input channel to be sent to the associated manager for processing. The default input manager for the Base Implementation is the Information Timer Based Buffer Input Manager. Assigning an output channel to a manager will result in any object sent to that manager to be written to the channel. The default Base Implementation output manager for information is the Information Timer-Based Buffer Output Manager. This buffer's operations are not unique to any specific Phoenix service and are suitable for all cases where transmission of information is done via output channels. The following diagram shows the table setup for configuring a CM with one input and one output manager along with some accompanying channels.

Output Channels

Channel Name	Channel Context
ac-channel	ChannelContext-1
ship-channel-1	ChannelContext-2
ship-channel-2	ChannelContext-3

Input Channels

Channel Name	Channel Context
in-channel	ChannelContext-0

Output Managers

Manager Name	Manager
Ship-Mgr	Info-Out-Mgr-1

Input Managers

Manager Name	Manager
In-Mgr	SS-Mgr-1

Channel-to-Manager Mappings

Manager Name	Channel Names
In-Mgr	In-channel
Ship-Mgr	ship-channel-1, ship-channel-2

Figure 7 - Channel Manager Configuration Tables

Figure 7 shows an example configuration of a Channel Manager's internal constructs. The manager contains a set of input and output channels, a set of input and output maanagers, and a set of channel to manager associations. Each channel and manager is required to have a unique name for mapping purposes. The name for a channel or manager is set by setting the name field of a channel's context or the name field of the manager (set through the manager's constructor). The example in Figure 7 shows how channels are assigned to a manager for managed input and output operations and it also shows that channels need not be assigned to managers at all, if unmanaged I/O is desired. The output channel named "ac-channel" is not assigned to an output manager and, hence, is an unmanaged output channel.

Service Multiplexor

Support for interactions with multiple service instances is provided by the service multiplexor interface. Service multiplexors are conditional multiplexors, i.e. they compute an output based on the given input and a set of rules that govern how the multiplexor functions. The output of a service multiplexor can be anything due to the use of the Java wild card type. However, it should be noted that the object invoking the service multiplexor must be capable of processing the result returned by the multiplexor.

The Base Implementation contains a Default Service Multiplexor (DSM) whose conditions are the type of objects it is provided. Currently, these include the Phoenix Information, Event, Information Type Context, Information Brokering Context, Information Query Context, and Event Notification Request Context interfaces. Supported outputs from the DSM include Information Timer-Based Buffers and channels and Phoenix service stubs. The following diagram shows a DSM for a Submission Service that is configured to write directly to output channels.

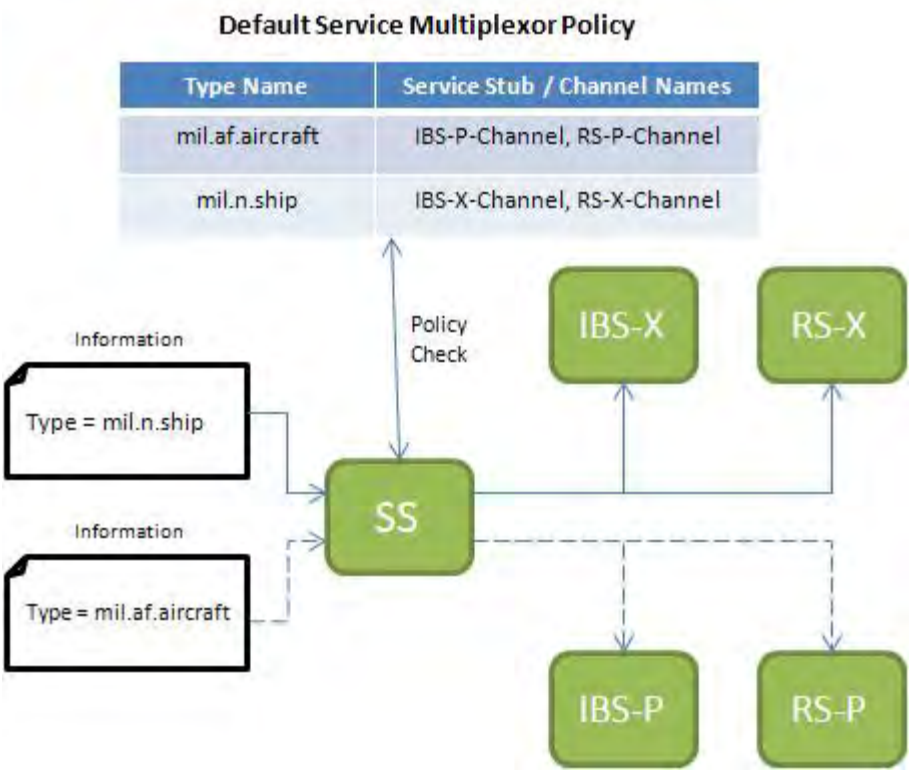


Figure 8 - Default Service Multiplexor Simple Example

IBS-P Information Brokering Service for Properties-based Information

IBS-X Information Brokering Service for XML-based Information

RS-P Repository Service for Properties-based Information

RS-X Repository Service for XML-based Information

SS Submission Service

The SS in this diagram is configured with a DSM that directs properties-based information to one set of services and XML-based information to another set of services. In this setup the SS is the invoker of the DSM and performs output channel write operations directly, with no service level buffering of outgoing information. The DSM can also be configured to support multiple types of returns for one or more conditions. The following diagram depicts this type of configuration and its resulting information flow.

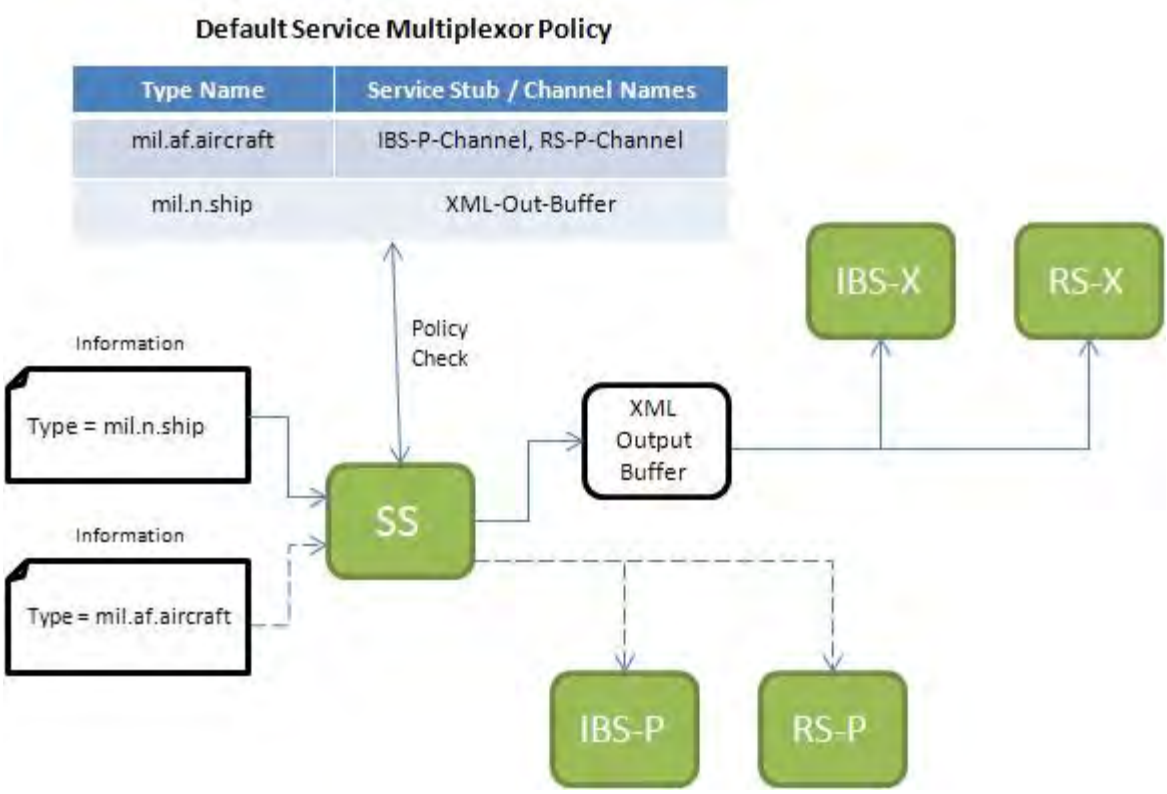


Figure 9 - Default Service Multiplexor Complex Example

IBS-P Information Brokering Service for Properties-based Information

IBS-X Information Brokering Service for XML-based Information

RS-P Repository Service for Properties-based Information

RS-X Repository Service for XML-based Information

SS Submission Service

This diagram shows a SS that is configured to add XML-based information to a specific output buffer but to directly write properties-based information to the configured set of output channels.

Most Phoenix services utilize an instance of the Default Service Multiplexor for configuring its interactions with other Phoenix services. This multiplexor forwards objects based upon the fully qualified object class name. For example, it may be configured to send all Information instances to service X and all event instances to service Y. There exists another multiplexor, the Information Service Multiplexor, that supports multiplexing of information by type name. For example, the Information Multiplexor may be configured to send all information of type 'mil.n.ship' to service A and all information of type 'mil.af.cot' to service B. This multiplexor only supports objects that implement the Information interface.

Some of the Base Implementation services have a Service Multiplexor setup at start-up time but do not currently use it for any operations. Examples of this include, but are not limited to, the Authorization and Session Management services.

Task Scheduler

The Base Implementation services use tasks and the task scheduler to perform their service-specific processing. The default implementation uses the Timer (java.util.Timer) and Timer Task (java.util.TimerTask) objects to implement tasks and the task scheduler. This capability is used by several services to schedule periodic status updates for various components including input and output channels and subscriptions. It is also utilized by the Dissemination Service as an integral part of its Consumer Black Listing capability.

Session

The session component package contains the classes used to describe actors and their respective user interactions with the Phoenix services in the form of contexts and the Session Track object. The Actor Context is used to describe who the actor is including security credentials or other data about the actor's identity. The Session Context is a service generated object that is used to track the actor's interactions with the Phoenix services; much like an HTTP session is used to track a user's online activity. The Session Track object is used by all Phoenix services to track what actors are making invocation calls on their exposed interfaces and for authorization operations within those invocations.

Session Identifier as an Object

A Java object is used to represent the session identifier to support any implementation of session identity consistent with the Phoenix architecture. Having this field be represented by a generic object allows one implementation of the services to use some form of a token while another implementation may use a simple string. A specific representation of this field has not been selected for the Phoenix Base Implementation.

Stream

The stream component package contains the streaming specific components shared amongst the set of streaming services. These components include the basic contexts and enumerations used by the Base Implementation to provide support for streaming operations.

Service Implementations

The Phoenix Base Implementation services can be organized into two distinct categories: Edge and Operational. Edge services are fully exposed to edge actors and may even be located within edge actor address spaces (i.e. they may be code that is downloaded and executed on an edge actor's machine). Operational services provide information management capabilities and are hosted by remote machines so they can be accessed as required by the Service Oriented Architecture (SOA) for information management operations.

Base Implementation Services are also broken into two functional categories: Administration and Information. Administrative services provide functionality that enables advanced information management operations (i.e. authentication and authorization or service brokering) while information services provide the basic functions for managing information (i.e. information type management and information brokering).

In addition to functional categories, the Base Implementation Services are also grouped by the expected interaction levels and deployment locations; both service to service and edge actor to

service. There are five of these groupings, identified in Figure 10: Edge Services, and Tier 1 through 4 Services. Edge services live within the address space of edge actors while tiered services may have edge facing connections but do not live within an edge actor's address space.

Figure 10 - Service Packages

AS Authorization Service

CRS Client Runtime Service

CS Connection Service

DS Dissemination Service

ENS Event Notification Service

FMS Filter Management Service

IBS Information Brokering Service

IDS Information Discovery Service

ITMS Information Type Management Service

QS Query Service

RS Repository Service

SBS Service Brokering Service

SMS Session Management Service

SS Submission Service

SUS Subscription Service

XBS Stream Brokering Service

XDS Stream Discovery Service

XRS Stream Repository Service

The following descriptions provide insight into the semantics and usage of the Base Implementation of the Phoenix Abstract Architecture. Other implementations of this architecture may choose to group their services differently and even define the services' operational

semantics differently.

Edge - Actor Services

The Base Implementation Actor Services are services that are fully exposed to the edge actors, meaning that they are available for both control and channel operations. Edge services may be hosted by edge actors within their own address space (i.e. integrated into a non-Phoenix application to provide connectiviy to Phoenix services). Actor Services include:

- Client Runtime Service

Client Runtime Service

This class ensures that there is a service oriented presence on the client-side to support event notification and connectors for reach-back from services to the client. This allows core IM Services the ability to influence external actors' address space providing a possible location for client -side policy enforcement and updating, event notification, or other service-to-external actor interactions. This ability becomes doubly important when operating on a disadvantaged network where actor communications may phase in and out over time due to networking degradation or other operational conditions. In this environment the client runtime service may provide a network buffer at the application level by queuing outgoing data until it can be transmitted or it may provide proxy IM capabilities for the client while it is disconnected from the network.

Tier 1 - Information Management Services

The Base Implementation Tier 1 Information Management Services directly interact with the edge actors via information and event channels. Control operations upon these services by edge actors are possible, depending upon the security policies being enforced by the implementation. Tier 1 services include:

- Dissemination Service
- Event Notification Service
- Submission Service

Dissemination Service

The Dissemination Service (DS) performs simple information distribution operations based on a round-robin scheduling algorithm. The DS is responsible for creating the information channel between a consumer of information and itself. This service is used by the Information Brokering and Repository Services to deliver information to registered subscribers and designated result set consumers, respectively. When an instance of information is read by the DS, it retrieves the list of channel definitions from the information instance's resident context, creates the channel(s) if they do not already exist, and then writes the information instance to each output channel in turn. It is important to note that no copying of the information instance is done during the dissemination process, the same instance of information is written to the output channel for each interested consumer, as shown by the figure below. For an in-depth Base Implementation

operational flow for information submission, brokering, and event notification or information delivery see the Submission Service section of this document.

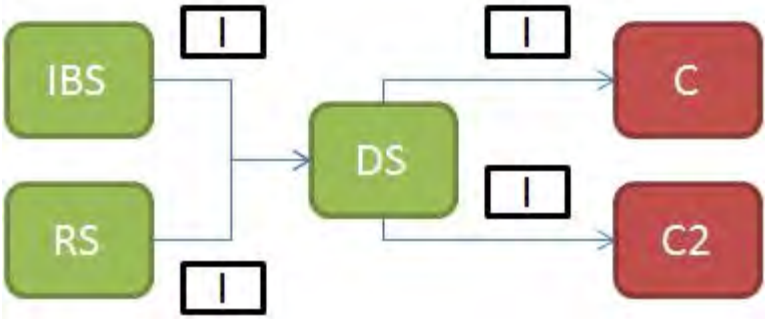


Figure 11 - Dissemination Service Use Cases

C	Consumer One	**I**	Phoenix Information instance
C2	Consumer Two	**IBS**	Phoenix Information Brokering Service
DS	Phoenix Dissemination Service	**RS**	Phoenix Repository Service

Black Listing

The Dissemination Service may be configured to track consumers that are no longer reachable and to attempt to contact them again at some point in the future. This capability is referred to as 'Black Listing.' When a delivery attempt is made, the DS will attempt to deliver the same object to the same consumer a configurable number of times. Once this limit is reached the DS will clean up the related output channel and add the consumer information to its Black List. Any further objects tagged for delivery to this consumer will not be delivered to them, nor will they be cached for later delivery. If configured to do so, after a set amount of time the DS will remove the consumer from its Black List. After this point the next object that is received that should be delivered to this consumer will result in the DS attempting to contact and deliver the object. Due to timing, visibility, and threading concerns this ability does not work very well with asynchronous deliveries at this time. By this we mean that if the max delivery attempts is set to three the DS may attempt to send more than three messages to a particular consumer before they are Black Listed resulting in lost computational cycles. This may be corrected in the future through the use of custom output channel handlers for the DS or via some other design.

Dissemination Service Events

The DS can be configured to periodically fire events that describe the current status of the registered consumer channels. The settings for these events are located in the service's context and are fully configurable. The settings for output channel status update event firings are as follow:

- *Event Firing Enabled* - The flag that globally enables or disables event firings for the DS.
- *Output Channel Updates Enabled* - The flag that enables output channel status event firings.

- *Output Channel Status Update Period* - The fixed interval, in milliseconds, between output channel status events. The Output Channel Status Event contains a copy of the channel context as a body.

Event Notification Service

The Event Notification Service (ENS) uses class and actor identifier matching algorithms to pair fired events with registered notification requests. The Base Implementation of the ENS does this using simple comparison operations over the firing actor identifiers and the fully qualified class names of the fired events. By providing methods for managing the internal event registry of the ENS, the Phoenix Architecture provides system engineers and developers a mechanism for creating and utilizing custom event classes. The only caveat to this is that these custom events must implement the Event interface.

Registering Event Descriptors

Registering an event descriptor with the ENS is simple. An actor invoking the registration method provides a sample instance of the event class being registered along with a human readable description of the event. The Base Implementation of the ENS will generate a unique identifier for the event class and store the sample instance, the registration identifier, and the provided description. If an event class has been previously registered, the Base Implementation of the ENS will return the registration identifier for the currently registered event.

Event Registration Identifiers

The registration identifiers for events are used by actors registering for event notifications. This is the field that provides the requesters the capability to define sub-sets of the registered events that they are specifically interested in. The Base Implementation of the ENS uses the fully qualified class name of the sample event instance provided at registration time to generate unique identifiers.

Use Case : Consumer Hit Lists

The current Use Case for the ENS in the Base Implementation is the notification of interested consumers for submitted information. In this case, the Producer of the information registers a subscription with the Information Brokering Service that requests a consumer hit list be generated for all matching information, instead of simply forwarding matching information to the dissemination service and back to the Producer. Figure 12 shows the interactions among actors when an Event Notification request for a consumer hit list is registered with the IBS.

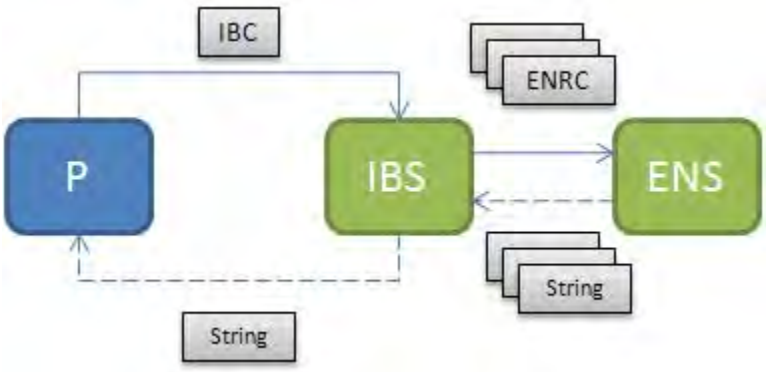


Figure 12 - Consumer Hit List Registration

E Phoenix Event instance

ENRC Phoenix Event Notification Request Context

IBC Phoenix Information Brokering Context

IBS Phoenix Information Brokering Service

P Producer

String The Event Notification request identifier and the Information subscription identifier.

The event notification request process is kicked off by the Producer submitting an Information Brokering Context to the IBS. This IBC contains the expression for information to match as well as designating the result of the brokering operations to be a consumer hit list event in place of forwarded information. The IBS recognizes this IBC as an event notification request and registers the request with the ENS. The resulting request identifier is returned to the IBS and the registered subscription identifier is returned to the Producer. The request identifier is used by the IBS to generate Events specific to the request.

After the Producer registers for consumer hit list notification, it begins submitting information to the Submission Service. The information is then forwarded to and brokered by the IBS. The IBS recognizes that one of the matching subscriptions for the information instance desires a consumer hit list in place of forwarded information and generates and fires an Event that contains the list of matching consumers. The list of consumers contains actual end point descriptions for out-of-band subscriptions and consumer identifiers for in-band consumers. The fired event is received by the ENS, matched to the corresponding notification request(s), and delivered to the event consumers, in this case the Producer. This process is depicted by the figure below.

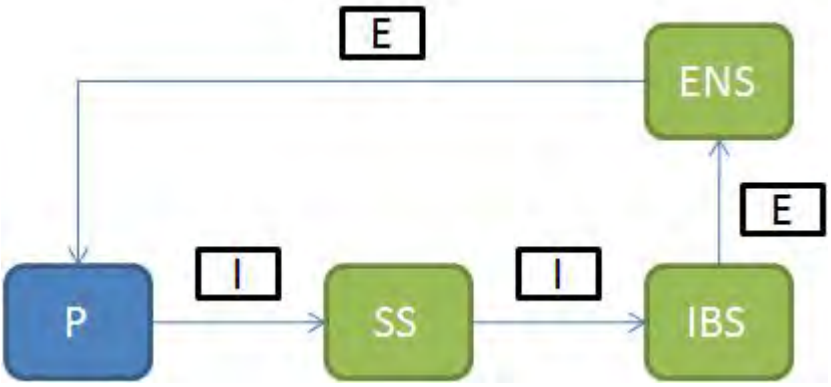


Figure 13 - Consumer Hit List Notification

E Phoenix Event instance

ENS Phoenix Event Notification Service

I Phoenix Information instance

IBS Phoenix Information Brokering Service

P Producer

SS Phoenix Submission Service

Base Implementation Event Classes

The table below provides the locations within the Base Implementation where services are using events and the ENS. This table associates pre-defined event classes with the service(s) that generate and fire the event instances.

Event Type	Location
Output Channel Status Event	Dissemination Service
Consumer Hit List Event	Information Brokering Service
Subscription Status Event	Information Brokering Service
Input Channel Status Event	Submission Service
InformationReceiptAckEvent	Client Runtime Service Submission Service

Submission Service

The Submission Service (SS) is designed to support the reception of information over Phoenix channels. The SS can host as many or as few input channels as physically possible by the hardware and software limitations placed upon it. The main duty of the SS is to forward information that is received to other IM services such as the Information Brokering Service (IBS) or the Repository Service (RS). The SS may forward information to any other information service, including other SS instances, based on the conditions defined by its Service Multiplexor policy. The Base Implementation SS may be optionally configured to perform information validation operations.

Forwarding of Submitted Information

The SS is designed to forward submitted information to other information processing services. The figure below shows the concept of information submission and forwarding to a single IBS and RS after the optional execution of information validation via type definition lookup on the Information Type Management Service (ITMS).

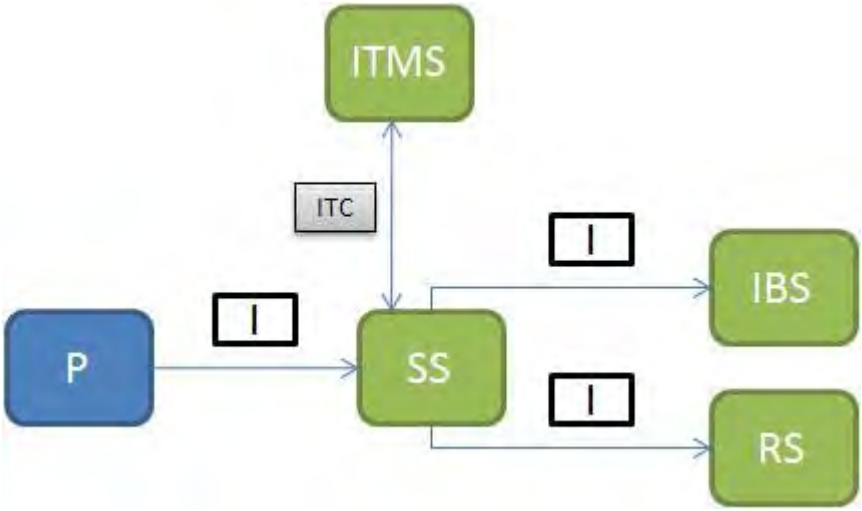


Figure 14 - Forwarding of Submitted Information

I Phoenix Information instance

IBS Phoenix Information Brokering Service

ITC Phoenix Information Type Context

ITMS Phoenix Information Type Management Service

P Producer

RS Phoenix Repository Service

SS Phoenix Submission Service

Submission Service as a Proxy

It is envisioned by the Base Implementation design team that this basic SS can be extended to form the core for an information submission proxy service. Such a proxy service would receive data in raw, non-Phoenix formats (such as Cursor-on-Target) and convert it to Phoenix information instances. This could also be achieved by a custom channel implementation used by the Base Implementation SS. The necessary design decisions regarding the submission, reception, and translation of non-Phoenix data to information are specific to each project utilizing the Phoenix Architecture and Base Implementation.

Submission Service Events

The SS can be configured to post status updates for its configured input channels at a regular interval. The settings for this are part of the service's context and are fully configurable via the service's Spring configuration file. The Service Multiplexor policy determines what events are sent to which Event Notification Service(s). The SS context settings related to input channel status update events are:

- *Event Firing Enabled* - The flag that globally enables or disables event firings for the SS.

- *Input Channel Status Updates Enabled* - The flag that enables input channel status event firings.
- *Input Channel Status Updates Period* - The fixed interval, in milliseconds, between input channel status events. Input channel status events contain a copy of the channel context for the input channel being reported on.

The SS can be configured to post information submission acknowledgment events when information is received via one of its input channels. The settings for this are part of the service's context and are also configurable via the service's Spring configuration file. The settings related to information submission acknowledgment events are:

- *Event Firing Enabled* - The flag that globally enables or disables event firings for the SS.
- *Information Receipt Ack Enabled* - The boolean flag that enables or disables information acknowledgment event firings.

Information receipt acknowledgment events contain an Information Ack object as a body. This object wraps the identifier for the information instance and the actor identifier of the producer, if available.

Information validation

Information validation is also performed on request by the SS. The SS context contains a mapping of information types and their associated validation mode. Validation modes for the base implementation are integers.

Validation Mode	Definition
0	Do not validate any instances of the associated information type.
1	Validate first instance seen of the associated information type.
2	Validate all instances of the associated information type.

When a validation is performed, the information type name is used to retrieve the type definition. Type definitions are retrieved once from the Information Type Management Service and then cached locally for the lifetime of the SS. Metadata and payload from the information instance being validated are processed if, and only if, the type definition contains a schema for each. If one field has a schema and the other does not, then only that one field with a schema is validated. If validation fails for one of the fields, a Validation Failed Exception is thrown back to the SS. This exception is reported in the log and the information instance is discarded. The discarded information is not sent to any information brokers or stored in any repository.

Tier 2 - Information Management Services

The Base Implementation's semantics do not allow information channel operations between edge actors and the Tier 2 Information Management services. These services either do not directly process information or they do not directly communicate with edge actors to receive or deliver information. This set of services includes:

- Connection Service
- Filter Management Service
- Information Brokering Service
- Information Discovery Service
- Information Type Management Service
- Query Service
- Stream Brokering Service
- Stream Discovery Service
- Subscription Service

Connection Service

The Connection Service (CS) performs complex data dissemination operations. Essentially this service acts as a stream dissemination service. Where the dissemination service performs simple, list-based data dissemination, the connection service uses pre-structured routes which are built previous to an object being received. The connection service allows registration of sources and sinks for data, and then allows for the multiplexing and de-multiplexing of sources to sinks. This service supports streaming behavior by allowing for faster dissemination of data without having to serialize or broker each individual object of information. The connection service can either be used directly if the producer and consumer have known, predefined, settings for their streams (Figure 15), or can be used via a proxy service, such as the Stream Brokering Service (Figure 16).

Figure 15 - Connection Service, Direct Use

The Stream Brokering Service will wrap the processes of the Connection Service and manage much of the complexity under the hood. The connection service has methods which allow direct manipulation of a connection group (the centers of data flow) and its membership (sources and consumers), and make matching data flowing from one source to another much simpler.

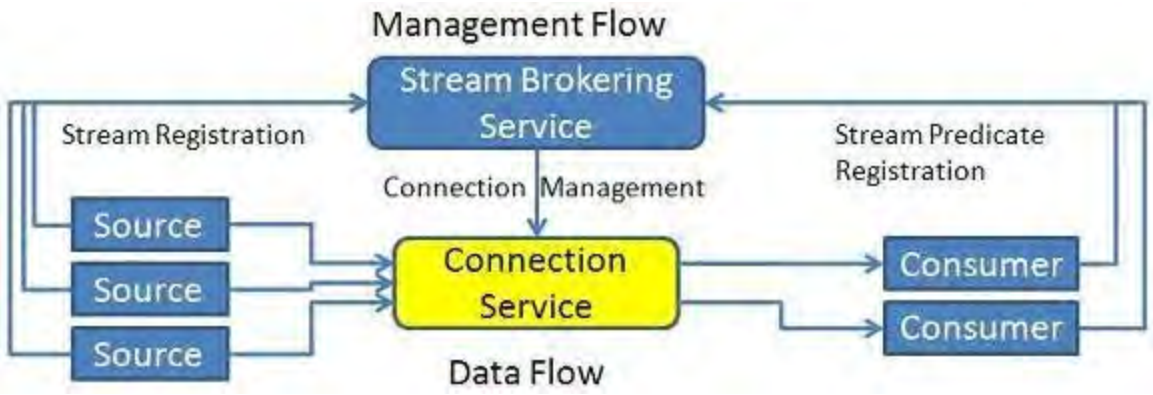


Figure 16 - Connection Service, Proxy Use

Filter Management Service

The Filter Management Service (FMS) is responsible for maintaining the registry of filters to be used by actors and for creating orchestrated chains of these filters for use by actors during filtering operations. An actor may register new filters or create a filter chain from previously registered filters.

Filter Registry

The filter chain registry maintained by the FMS is a simple map structure that uses the filter context names as its keys. The values are the example instances of the filter classes. Each filter class must implement both a default, no argument constructor and a constructor that accepts a Filter Context as its only argument.

Creating a Filter Chain

Currently an actor creates a filter chain by creating a Filter Chain Context which contains the desired input and output types for the chain. The FMS will then scan its registry and attempt to assemble a filter chain that will result in the provided output type. There is no current interface method for setting the desired number of filters in the chain or any other specific attributes for creating the filter chain, making this a very fundamental capability. Future work may include upgrading the filter chain creation abilities offered to external actors.

Information Brokering Service

The Information Brokering Service (IBS) uses a pluggable architecture to support the set of potential expression processor technologies. The actual processing code, specific to the information formats and technologies used for processing is all contained within the expression processing package, while the supporting code that loads and executes it is contained within the IBS.

The Base Implementation of the IBS currently supports integration with other services via the Service Multiplexor. This allows the IBS to forward brokered information to other services of any type. The most common example of this would be to forward brokered information directly to one or more Dissemination Services for delivery to interested consumers.

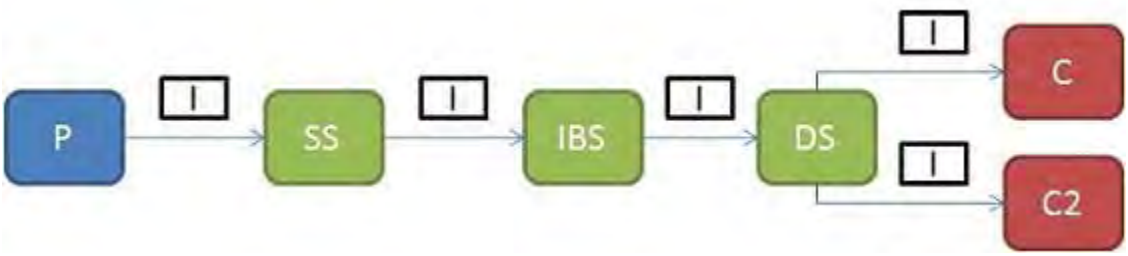


Figure 17 - Information Brokering Service

C Consumer One

C2 Consumer Two

DS Phoenix Dissemination Service

I Phoenix Information instance

IBS Phoenix Information Brokering Service

P Producer

SS Phoenix Submission Service

Figure 17 shows the flow of information through the IBS. Producers submit information via one or more Submission Service instances. These SS instances forward the information to the IBS for brokering. The IBS brokers the information, and as a result, tags each information instance with a list of consumer channel definitions associated with the expressions that matched the information. The IBS then forwards the information instance to a Dissemination Service for delivery.

Subscription Registration

To register a subscription a consumer sends a Subscription Context by invoking the 'registerSubscriptions()' method on the IBS. This method is inherited from the Subscription Service (SUS) interface. This context contains the channel contexts that define the output channels to be used to disseminate matching information instances to consumers. It is these channel contexts that the IBS tags brokered information with. The IBS completes the subscription process by registering the expressions contained within the Subscription Context with its resident expression processors. This registration is performed based on expression type, each processor supporting a single, unique expression type.

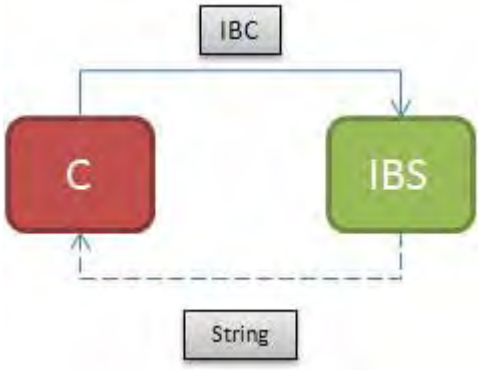


Figure 18 - Information Subscription

C Consumer

IBC Phoenix Information Brokering Context instance

IBS Phoenix Information Brokering Service

String A java.lang.String instance, i.e. some flavor of identifier be it subscription or consumer.

Figure 19 depicts the operational flow for subscription registration.

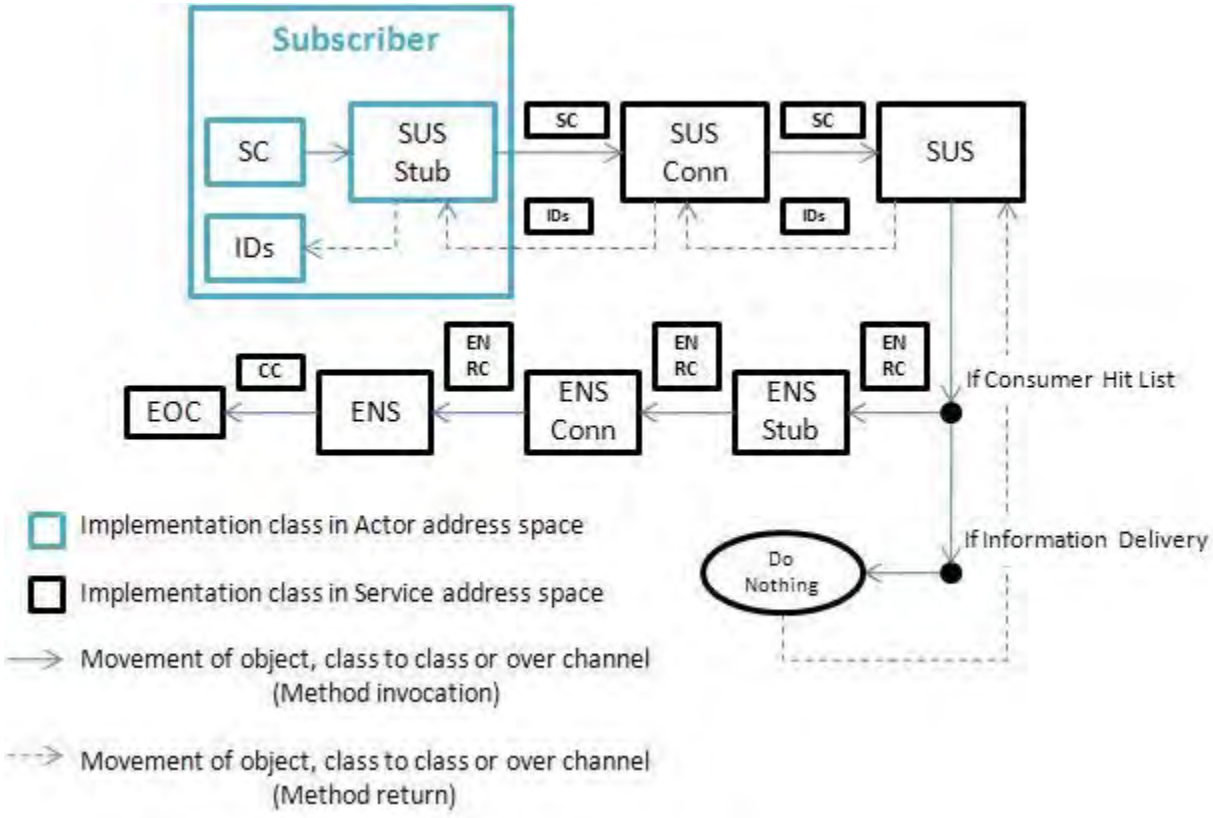


Figure 19 - Subscription Operational Flow

CC Channel Context

Conn Connector

ENRC Event Notification Request Context

ENS Event Notification Service

IDs Subscription Identifiers

SC Subscription Context

SUS Subscription Service

The subscriber begins the subscription registration process by creating a Subscription Context. This context is submitted to the Subscription Service via a control channel (the service connector & stub). In this example, the Subscription Service is also its own Information Brokering Service so the service checks the brokering result type for the submitted subscription. If it is set to consumer hit lists an Event Notification Request Context is generated and registered with the configured Event Notification Service(s) through another control channel. Otherwise the result

type is set to information so no additional actions are required at registration time. The subscription context is stored by the IBS and its expressions registered with the appropriate expression processors in preparation for matching against incoming information.

Subscription Identifiers

Registered subscriptions are identified by their resident names. These names are part of the subscription context. The default value for a context name is the context identifier guaranteeing that, if no name is set, the identifier is unique.

Subscription Evaluation

Subscription evaluation is done via the supporting expression processors. The IBS evaluates an information instance with each processor, appending the returned subscription identifiers to a master list of identifiers. When the evaluations are complete the IBS then loops through the registered subscriptions and checks if the number of expressions for a subscription equals the number of times that subscription identifier appears in the master list. If it does then the subscription matches the brokered information and the correct information brokering operations will take place depending on the defined result for that subscription (forward information to consumers or event notification). If the subscription identifier does not appear in the master list the requisite number of times then the subscription does not match the brokered information and nothing is done.

Updating Expressions

The Phoenix Base Implementation will only support the updating of expression tests or other light weight context values. It will NOT support the updating of registered consumer channels for a subscription. Changing these (adding, deleting, etc.) or their resident static settings (host name, port number, etc.) will require a drop and re-registration process.

Events

The IBS provides the capability to register for consumer hit lists for subscriptions instead of receiving matching information instances. These Consumer Hit Lists are delivered via custom Consumer Hit List Events whose bodies are a Consumer Hit List object.

The IBS can be configured to fire subscription status events at a fixed rate by using settings located in its service context. These settings are fully configurable via the Spring configuration file, when using one. The settings for firing subscription status update events are as follow:

- *Event Firing Enabled* - The flag that globally enables or disables event firings for the IBS.
- *Subscription Heartbeat* - The flag that enables or disables subscription status event firings.
- *Subscription Heartbeat Period* - The fixed amount of time, in milliseconds, between subscription status events.

Subscription status events are used to report the current status of individual subscriptions. The body for this type of event is a Subscription Status object that contains the identifier of the information brokering context that was registered, the subscription that was generated from that context by the IBS, a boolean flag denoting whether or not the subscription is suspended, the

total number of information instances processed against the subscription to date, and the time stamp of when the last matching message was processed.

Information Discovery Service

The Information Discovery Service (IDS) provides a simple interface for "discovering" what information types are known to the Information Management (IM) services and what services are supporting which types.

Discovering Information Types

The IDS communicates with one or more Information Type Management Services (ITMS) to provide the information type discovery capability. This allows actors a central focal point for finding out what types of information are known to the set of information management services. The reliance on the ITMS (one or more) means that unregistered types of information are not discoverable by the IDS.

Discovering Supporting Services

This capability allows an actor to discover what services are available to support information of a specific type. This capability also offers the option to search for specific service types. If an actor wishes to submit information of a certain type through a service that has been deployed for this function they can use this capability to locate a stub for said service, allowing them to connect to it and submit their information. This capability has been implemented using the Service Brokering Service (SBS) and its service descriptor and brokering capabilities. Therefore a service is not truly "discoverable" by the IDS unless it has been registered with the SBS (one or more) that the IDS is communicating with.

Information Type Management Service

The Information Type Management Service (ITMS) stores the definitions for registered information types. The information type name is also referred to as the information type identifier. The ITMS stores the information type definitions in memory using a simple java.util.HashMap construct where the key is the information type identifier and the value is the actual Information Type Context object. This map of information type definitions is included as part of the ITMS service context and, as such, is included in any store or load operations performed by the ITMS. The ITMS also supports the concept of archiving information type definitions, meaning that the definition is not considered active, but is kept for archival and tracking purposes. This archive function is supported via the XStream library just as the store function is for all services.

Information Type Definitions

Information type definitions must contain an information type identifier. Un-typed information is supported by the definition of an un-typed information type where the information type identifier is either set to the empty string or null. Registered information types do NOT, however, require metadata or payload schemas. A registered information type definition may contain one or both schemas in an effort to describe the format and content of a specific type of information.

Information Type Relationships

Currently the Base Implementation does not support any notion of relationships amongst information type definitions. A parent-child relationship scheme, with children inheriting and augmenting the metadata definitions of their predecessors may be implemented sometime in the future.

Schemas

Information type definitions may contain a metadata or payload schema or both. These schemas are represented in the abstract architecture as Objects to allow flexibility at implementation time regarding what exactly these definition documents are to be. For XML information a schema document may be a DTD, XML Schema Document (XSD), or something else. For the home-grown support for name-value pair based metadata the schema document is an Attribute Schema Document (ASD). The ASD format is an AFRL developed definition of name-value pairs, their data types, and expectancy (REQUIRED vs. OPTIONAL).

Auxiliary Elements

Since information type definitions are context based they can support any additional elements other than type identifier and metadata and payload schemas. For example information type definition contexts could be used to store data processing libraries specific to an information type or other useful items such as priority tags. The Base Implementation does not exercise this ability, but it exists and needs to be advertised.

Example Information Types

The Base Implementation uses three simple example information types for basic testing of the implementation's information management capabilities. Their information type identifiers are: *mil.af.aircraft*, *mil.n.ship*, and *mil.a.infantry*. These three information types provide support for two unique representations of metadata: name-value pair and XML. The *mil.af.aircraft* information type has metadata that is name-value pair based while the other two types have XML based metadata.

Information Type Identifier	Metadata Schema	Payload Schema
mil.a.infantry	mil.a.infantry Metadata.xsd	None.
mil.af.aircraft	mil.af.aircraft Metadata.asd	None.
mil.n.ship	mil.n.ship Metadata.xsd	None.

Orchestration with Repository Service

The Base Implementation of the ITMS includes the ability to orchestrate with one or more Repository Services (RS). The ITMS, at start up, information type definition registration, and information type de-registration time will attempt to contact each of its configured RS instances to alert them to start or stop storing information of certain types.

At start up time, the ITMS will automatically attempt to load a saved ITMS state file. If one is not found, a warning is logged, but no errors occur. If one is found, the ITMS will set its state to that pulled from the file and tell each of its configured RS instances (also pulled from the state file) to begin storing information of the types loaded from the ITMS state file.

At information type definition registration time the ITMS will attempt to tell each of its configured RS instances to begin storing information of the types being registered. The Base Implementation of the ITMS blindly invokes this method call on each of the configured RS instances, without any special algorithm(s) to determine which should and should not store information of a specific type.

The case of information type de-registration is similar to that of registration, with the main action reversed. Instead of telling each RS instance to begin storing information of a specific type or types, the ITMS now tells each RS instance to stop storing information of that type or types.

Query Service

The Query Service (QS) currently supports synchronous or asynchronous query operations against one other QS. It is important to note here that all Repository Services (and their connectors and stubs) all extend the Query Service interface (and its connector and stub interfaces respectively) so that all RS instances are indeed QS instances as well and therefore able to be queried via the basic implementation of the QS. Future plans include supporting more than a single QS to fan queries out to, injecting some form of access control logic specific to query apportionment and processing, and support for result set aggregation, normalization (sorting) and intersection (eliminating duplicate results).

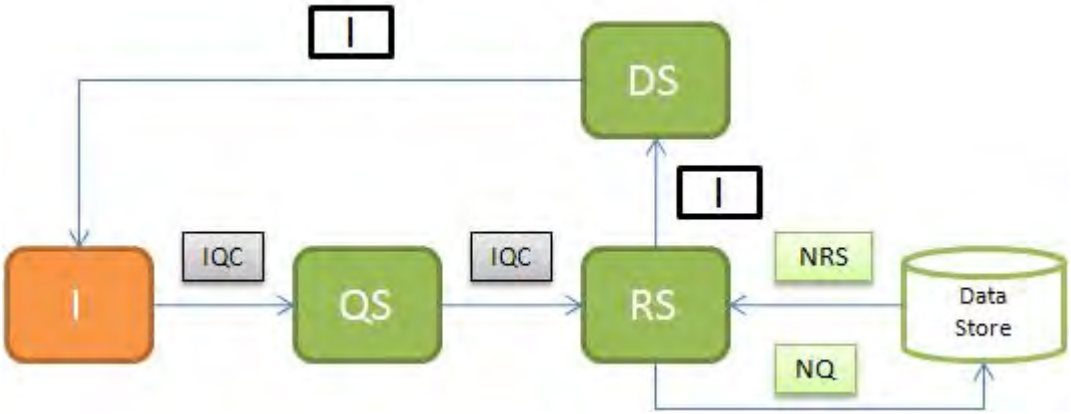


Figure 20 - Query Service

DS Phoenix Dissemination Service

I Phoenix Information instance

IQC Phoenix Information Query Context

NQ Native Query Statement (i.e. SQL, XQuery, etc.)

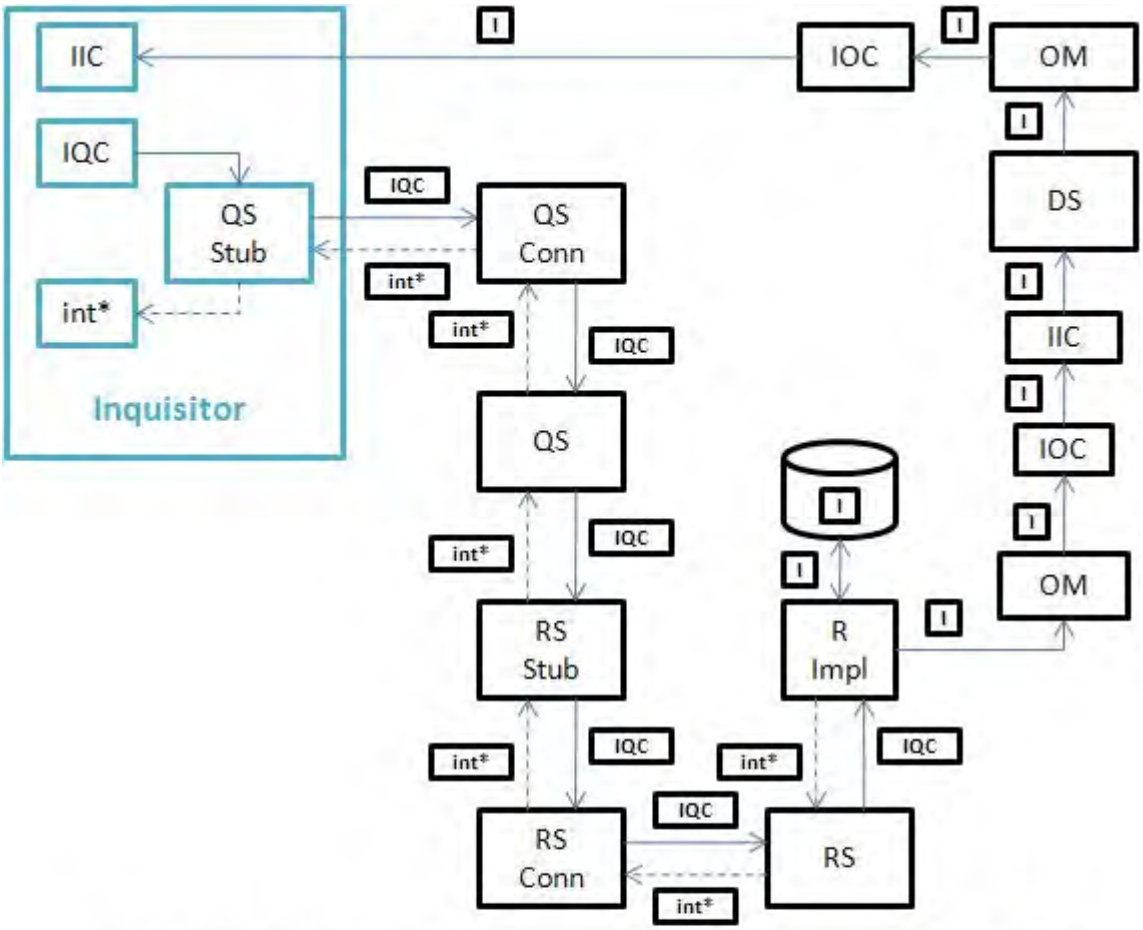
NRS Native Result Set (i.e. JDBC Result Set object, Berkeley Result object, etc.)

QS Phoenix Query Service

RS Phoenix Repository Service

Figure 20 shows a simple view of the interactions and data flows that occur during an information query operation. Briefly, an inquisitor submits an Information Query Context instance to a QS. The QS passes the IQC off to its resident RS who turns the IQC into a Native Query Statement. This NQS, tailored specifically for the underlying data store, is submitted to said data store and results in a Native Result Set object being returned to the RS. The RS converts the NRS object into a set of Phoenix information instances that it then sends to the DS for delivery to all identified query consumers, in this case only the original inquisitor.

Figure 21 depicts the operations required for a Phoenix query to execute and deliver its result set to a consumer. This example assumes that the actor issuing the query is also the only consumer of the result set.



* - Size of result set only returned when query is executed in synchronous mode.

Implementation class in Actor address space

Implementation class in Service address space

Movement of object, class to class or over channel (Method invocation)

Movement of object, class to class or over channel (Method return)

Repository (File, Berkeley, etc.)

Figure 21 - Query Execution

Conn Connector

DS Phoenix Dissemination Service

I Phoenix Information instance

Impl Implementation

int The result set size.

OM Output Manager

IIC	Information Input Channel	**QS**	Phoenix Query Service
IOC	Information Output Channel	**RS**	Phoenix Repository Service
IQC	Phoenix Information Query Context		

An actor begins a query operation by constructing an Information Query Context and submitting it to a Query Service via a control channel (connector & stub). The Query Service will forward this context to applicable Repository Services based on its interior implementation logic or policy. The Repository Service then submits the query to one or more of its configured repository implementations for execution. Again, this decision is based on the service's implementation logic or policy. If executing a synchronous query, the repository implementation(s) will return a result set size to the Repository Service, which will aggregate the result set sizes of all applicable repository implementations and return that number to the Query Service. The Query Service will likewise aggregate the result set sizes of all applicable Repository Services and return that total size to the issuing actor as the total result set size. If the query was executed in asynchronous mode, the repository implementation will return zero for a result set size as soon as the query has been submitted for execution. The same chain of aggregation applies but the result is always zero and the query (or queries) may or may not be completed when the zero value is returned to the issuing actor. When the query has completed, the repository implementation retrieves the result set one information instance at a time. Each information instance is stamped with the identifier of the query and then sent to the Dissemination Service for delivery to the query consumers, in this case the actor issuing the query.

Querying Data Stores

Current repository interface implementations support queries over single information types, but not over sets of zero or more than one. A query with a null or unspecified expression will return all information instances of the designated type. Future work will include the abilities to query over all information types by not specifying a subset of types to apply the expression to and to specify specific sub-sets of information types to apply the expression to.

Stream Brokering Service

This Service offers the capability of registering producers as stream sources, and consumers as stream sinks. The Stream Brokering Service (XBS) wraps the functionality of Connection Service with Stream administration. Membership of streams, including their publication source's metadata and identity, consumers using stream-based expressions, and the registration of the stream definition itself, are all part of the Stream paradigm for this service. Data does not flow through this service, data is routed through a connection service, but the control and management operations for a stream are administered through this service.

This service is used to simplify the operations of stream management, as well as the intricacies of the connection service, by providing a wrapper which does much of the work for the producer and consumer in terms of stream registration being converted to a connection group definition, a consumer and source being registered and added as members of that connection group based on the stream expression or their stream Id. The main purpose of this service however, and why it is typically used, is because it uses the ontological terms and language of streaming so that the operations of the connection service can be much easier related to, and also specialized for streaming functionality. The stream channel types allowed are information, frame, and byte.

To see more specifics on this service and its methods, please consult with the documentation for the main stream brokering service interface. StreamBrokeringService

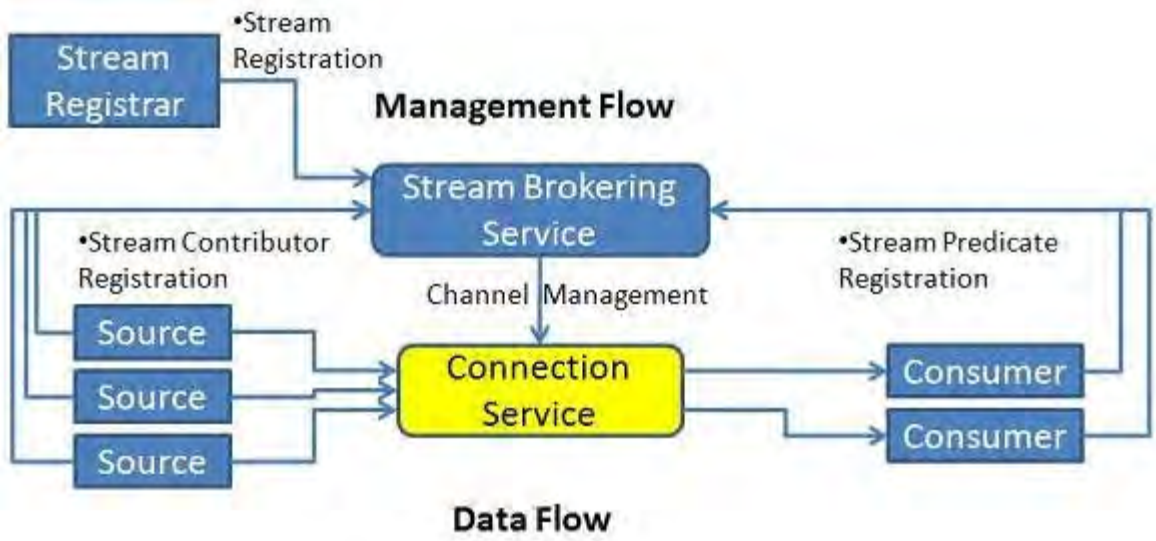


Figure 22 - Stream Brokering Service Use

Stream Discovery Service

This Service offers the capability of registering and querying streams. A Stream is an abstraction of a set of data, which could be in many formats (e.g. Frame, Information, Byte, etc.) A Stream which is registered may have an inactive or active state, and is stored with the stubs of services which are relevant to accessing or joining the Stream. The Stream queried for will be brokered upon immediately. Eventually there may be a capability to be notified of stream activations/registrations via an Event Notification mechanism. The Stream Discovery Service (XDS) allows for multiple expression processors, so Services could be registered as property-based, XPath, or other Stream Discovery Contexts, and then found when others perform a query using that same expression processor for which the Stream metadata was formatted.

The Stream Discovery Service is initially very similar to a UDDI registry, where an uploaded entry has a name, description, and metadata, associated with it, which queries are matched against, with results returning more useful information, such as finding the location of the repository where the Stream is stored, if it was stored at the time of its publication. A Stream may be registered with a variety of attributes, including the channel type which the data will conform to, the metadata for the stream (which all data published to the stream must comply with) and other context attributes.

To see more specifics on this service and its methods, please consult with the documentation for the main stream brokering service interface. StreamDiscoveryService

Subscription Service

The Subscription Service (SUS) is a Tier 2 Information Management service that provides a subscription registration interface to actors, both edge and internal. Subscription registration may be local or remote. Multiple SUS instances may be registered with another SUS, allowing for operations such as load balancing and subscription fan-out. The Subscription Service relies upon its internal Service Multiplexor for determining what subscriptions are registered with which

other SUS instances known to it, if any. Figure 23 shows a simple depiction of the two SUS configuration options.

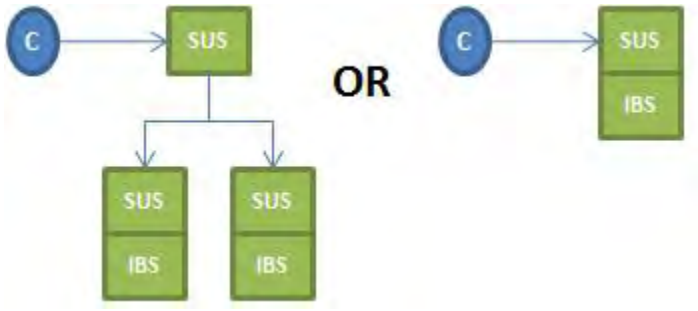


Figure 23 - Subscription Service Use

C Consumer

IBS Information Brokering Service

SUS Subscription Service

Tier 3 - Information Management Services

Tier 3 services provide basic information management functionalities, but provide interfaces that are not exposed to edge actors. These services are available only through other service proxies and are not intended to provide direct channel support to the edge actors. Tier 3 services are:

- Repository Service
- Stream Repository Service

Repository Service

The Repository Service (RS) has been implemented as a configurable class that can support multiple data stores at the same time, using the same code. A Repository Interface has been defined that describes the standard interface for interacting with a data store. This interface is used by the RS as the transparent facade for all data stores making the RS code 100% reusable amongst data store technologies, assuming that an implementation exists for the data store technology that conforms to the Repository Interface. i.e. An implementation of a Repository must exist that supports Berkeley DB XML if the RS is going to support storing and retrieving information in and from a Berkeley DB XML data store.

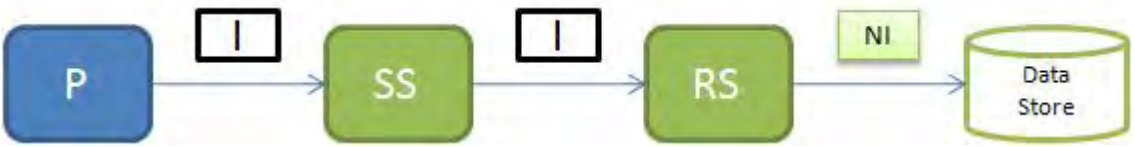


Figure 24 - Repository Service

I Phoenix Information instance **RS** Phoenix Repository Service

NI Native Insert Statement (i.e. SQL, etc.) **SS** Phoenix Submission Service

P Producer

Configuring the Repository Service

The Repository Service has been designed to be configured one of two ways, either as a standalone service or as part of a service orchestration with one or more Information Type Management Services (ITMS). Both of these configuration options require that the RS receive a complete copy of the information type definition to ensure that the underlying data store(s) create the corresponding data table structures correctly. For example, a database (RDBMS, NXD, etc.) requires the schema for metadata for an information type so that it can create the tables with correctly formatted rows and columns (or their equivalent) based on metadata field type (integer, string, etc.).

The stand alone service configuration contains a list of complete information type definitions for the information types to be stored by the RS. This configuration may be used in concert with the service orchestration configuration if desired. Stand alone RS instances are configured through the use of Spring configuration files. Stand alone configuration options do not supersede the service's ability to load an actual service context, containing a raw copy of service state. If a service state file exists, and is explicitly loaded by the RS or its service container, that saved state will override the configuration settings provided by the Spring configuration file. It is important to note here that the Base Implementation of the RS does NOT automatically attempt to load a service state file. The service container or some other external mechanism to the Base Implementation RS class must configure the service instance correctly and explicitly invoke the load function after service creation and prior to service start up.

The service orchestration configuration depends on the ITMS instances telling the RS which information types to store via connector and stub invocations. This configuration method is preferred for SOA deployments such as Enterprise Service Bus (ESB) or federated information spaces. Changes made to the RS configuration, i.e. what types it is storing, will supersede any that were loaded from a service state file. This is due to the fact that configuration via service orchestration requires that all services be instantiated AND successfully started prior to orchestration. This configuration option also dictates that all RS instances being orchestrated need to be started before the available ITMS instances. This is required because, on start up, the ITMS instances will attempt to load any pre-configured type definitions. If any are found each ITMS will attempt to invoke a control method on its known RS instances to tell each to begin storing information of those types. See the Information Type Management Service, Orchestration with Repository Service section for more details.

Archiving Persisted Information

Persisted information may be archived by invoking a specific method on the RS interface. The act of archiving information will pull persisted information out of the repository, write it to a set of output channels whose end points are the archival services or systems (i.e. the services or systems that will perform the actual archiving of the information), and delete the archived information from the repository. Archiving of information is initiated via the use of an instance of the Information Query Context interface, allowing for great flexibility to be exposed to the actors invoking the archive operation. Part of this context is a set of channel contexts that define the query consumer channels. These channels are the previously mentioned set of output channels

for the archival services or systems. These services or systems may, but are not required to, include another Phoenix RS. If no consumer channels are defined for the query context of an archive operation the Base Implementation RS will assume local archival operations and add a custom output channel that is basically a channel wrapper for a File Repository instance. File Repository was chosen as the default information archive technology because it uses XStream to write and read information instances to and from XML files in a loss-less fashion. The use of XStream makes the saved information instances agnostic and able to be read by XStream or any other XML processing application. If not using XStream, some development must be done to interpret the received XML, but no interpretation of third party proprietary formats or anything of the like need to be done.

Repository Interface

The Repository Interface standardizes the methodology for interacting with underlying data stores. It allows the RS to interact with disparate technologies, such as Native XML Databases (NXD) and Relational Database Management Systems (RDBMS), in a transparent manner. This interface defines methods for creating and destroying and writing to and reading from collections of information. The current set of supported data store technologies is as follows:

- File Repository
- Typed File Repository
- Berkeley DB XML Repository
- Mongo DB Document-Oriented Repository

File Repository

The simple file repository is a wrapper for straight file input and output operations using either the Local or a Network File System (LFS or NFS, respectively). Information instances are all stored in a single collection that maps to a set of file system folders. The File Repository is initialized to contain a single folder named "0" that can contain a configurable number of files. Each information instance is a single file. Once the maximum number of files is reached for a folder, the File Repository automatically creates a new folder, numbered in sequence (i.e. 0, 1, 2, 3...) and starts storing information instances there. This repository can continue to operate until the memory space afforded by the disk drive is exhausted. Information instance files are named using the instances information context identifier suffixed with a ".xml" extension. Information instance objects are converted to an XML representation by the XStream library. The File Repository does support the deletion of specific information instances, identified by information context identifier. Since the search mechanism (described below) is rather simplistic, delete operations can be expected to consume CPU time in direct proportion to the total number of records being stored by the repository.

The File Repository supports a very simple query language: context identifier matching. This means that to query a File Repository you must supply a set of information context identifiers. The File Repository will then search its folders for a file with that name. Since there is no directory of file names mapped to paths and no notion of typed information collections, searches of File Repository data stores can be expected to rise proportionately with the total number of records stored by the repository. Attempting to retrieve the number of records from a File Repository will return the total number of records being maintained, but since typed collections are not supported, no break down by individual information type is provided.

Since the File Repository is LFS and NFS based, it inherits the strengths, weaknesses, and limitations inherent to the hardware and software the file share is based upon. For instance, the

maximum number of files allowed in any single directory is directly dictated by the file share's controlling Operating System and any policies placed upon it.

Typed File Repository

The Typed File Repository extends the simple File Repository and adds support for typed collections of information. As such, query and deletion performance should be improved over that of the simple file repository, providing that the correct parameters are provided in the operational context to scope the query or delete operation to a sub-set of information types.

Berkeley DB XML Repository

The Berkeley DB XML Repository, hereafter referred to as the Berkeley Repository, provides an interface to the Berkeley NXD. This repository supports the full range of XPath and XQuery operations supported by the Berkeley database. While Berkeley DB XML supports applying XQuery over multiple collections, only XPath and XQuery expressions over a single database collection are supported by the Base Implementation. For the foreseeable future, any query submitted to a Berkeley Repository must be typed with a single information type and all other queries will result in a thrown exception. The current supported version of the Berkeley DB XML Database is 2.5.13. Berkeley DB XML is an Oracle product built on top of the Berkeley DB name-value pair database. Berkeley DB XML is a native C++ library with a Java bridge also provided by Oracle as part of the Berkeley DB XML distribution.

Berkeley DB XML has been shown in the past to be buggy and crash-prone for hazy or indeterminate reasons somehow related to memory management and allocation. The Berkeley Repository obviously inherits these problems wholesale and use of this repository should be limited to research and development efforts. Special care must be taken to ensure that the DB_CONFIG file does not contain values that require a larger portion of memory than there is available. Cryptic technical error messages add to the burden of dealing with this database as well.

One should ask, what with all these issues and concerns, why do we provide a Berkeley DB XML repository implementation? The answer is simple: It was the best Open Source NXD solution available when the Base Implementation repository technologies were evaluated. Others, such as Sedna and eXist are available as Open Source, but neither matches the shaky robustness of Berkeley when dealing with collections of information. Neither Sedna nor eXist support as many XML, XPath, and XQuery related features as Berkeley DB XML either. In addition Sedna, while Open Source and possibly a little more robust than Berkeley DB XML, is Russian product, making it virtually impossible to run on an accredited, or even non-accredited, government network.

Berkeley DB XML Performance and Fault Tolerance

Several configurable performance measures have been incorporated into the Berkely DB XML implementation of the Repository Service. To mitigate database corruption, an internal write buffer has been integrated to temporarily store information on disk and commit that information to the appropriate Berkeley repository when a buffer threshold is reached or a predetermined time limit has been exceeded. Both the save rate and the time to live for the buffer are configured through the BerkeleyRepositoryContext (via the get/setSaveRate() and get/setWriteTimeout() methods).

The Berkeley repository implementation has been optimized to handle larger repositories for a given information type. As the number of messages for a given information type exceed a predetermined threshold, the Berkeley repository will create a new container for the information

type. The threshold that instructs the Berkeley repository to create a new container is managed through the BerkeleyRepositoryContext via the get/setMultipleContainerThreshold. This threshold should be adjusted based on message size and frequency.

Berkeley DB XML Configuration

The Berkeley DB XML database, hereafter referred to as BDBXML, is configured by the settings stored in a file named "DB_CONFIG" which is required to be located in the home directory of the BDBXML instance being run. Each BDBXML instance has a configurable home directory that it uses to store its required files and folders, the DB_CONFIG configuration file being one of these. If no DB_CONFIG file is located the hard coded default settings are used, which often result in problems at runtime.

The most common use of the DB_CONFIG file is to set the amount of physical memory allocated to Berkeley DB XML. This is configured using the 'set_cachesize' entry in the configuration file. This setting requires three parameters: the number of gigabytes allocated, the number of bytes allocated, and the number of cache segments to break the allocated memory into. An example entry is shown below.

set_cachesize 0 268435456 1

Where the number of gigabytes is set to zero, the number of bytes is set to the equivalent of 256MB, and the number of cache segments is set to one. The next most common set of configuration settings involve locks, lockers, and lock objects. "A 'locker' is defined by Berkeley as a kind of database management container. A locker owns 'locks' which enact physical locks on pages of the database. Lockers are associated with containers, which own database handles, as well as documents, which may own cursors." [1] An example entry set for lockers, locks, and lock objects is as follows:

set_lk_max_lockers 20000

set_lk_max_locks 20000

set_lk_max_objects 20000

For more on lockers, locks, and lock objects reference the Berkeley Overview.

The Berkeley repository implementation comes with a few caveats though.

1. Berkeley DBXML inserts information very quickly, even while indexing multiple fields of varying types, but query is not nearly as optimized. Queries can typically take minutes to return a result set, even when constrained to indexed fields.
2. The Berkeley Repository is known to cause a total freeze of all VM operations when run as part of a suite of services deployed in a Java Service Container (JSC).

Mongo Repository

MongoDB is a document-oriented NoSQL database that allows users to store and retrieve unstructured information without requiring a database schema. NoSQL data stores are highly flexible, like a XML repository, while providing functionality that is commonly found in traditional

RDBMS systems. The MongoDB implementation adheres to the NoSQL concepts while focusing on document-based information while providing distributed storage and retrieval of information.

Running the Daemon

In order for the MongoDB repository implementation to function properly, a daemon process must be running. The daemon process is a native service (binaries exist for Windows, Linux, OS X, and Solaris) that can be configured to run in a distributed fashion. The daemon process can be installed as an OS level service or it can be run in place by invoking the `mongod` executable.

Storing Information

The MongoDB repository service expects to receive XML messages as the information to be stored. The XML is converted into a JSON like structure and stored in MongoDB. The conversion from XML to JSON is generic and as a result the conversion process does not delineate between numeric and string values. It is possible to store any type of unstructured information.

Querying Information

Queries (i.e., experessions) are formed using a dot notation to traverse a document heirarchy. An XML snippet is listed below to demonstrate the MongoDB query syntax:

```
<x>
 <y>
  <z>
   Some Text
  </z>
 </y>
</x>
```

Citing the XML above, if a user wanted to query for the phrase "Some Text" the following dot notation query would be created: `x.y.z = "Some Text"`. With respect to the repository service, the predicate within the InformationQueryContextInterface must be stored as a BasicDBObject via the `setQueryObject()`. For more information on the construction of a MongoDB query refer to the MongoDB Java Tutorial located at: http://www.mongodb.org/display/DOCS/Java+Tutorial

Deleting Information

The semantics for querying information and deleting information are the same.

Known Limitations

The converter that transforms XML to a JSON structure cannot distinguish numeric from alphanumeric data. This prohibits the use of numeric based operators (e.g., <, >, etc) since all data is treated as alphanumeric.

Future Work

Devise a sophisticated XML to JSON conversion process that is intelligent enough to determine if an attribute or node value is numeric vs. alphanumeric. Additionally, testing should be done to explore the possibilities of running the MongoDB daemon in a distributed fashion.

Stream Repository Service

The Stream Repository Service inserts frames into its associated data store(s). Although these interfaces are consistent with the general repository service, some methods are not functional, as the stream repository service is based on frames rather than information, which means that a) there is no type associated with the data, although it is correlated with the type associated with the stream context, b) there are no contexts for the frames, and c) frames are based around streams, which means that the repository should have streams as representative of the types of data, encapsulating methods which can retrieve the schema definitions for the stream, metadata content, and payload content. There is no actual insert frames method defined as part of the service API. Instead, the Repository Service receives frames over frame channels, which it reads internally, making insertion an internal process. This decision was made to ensure the physical separation of control versus data interactions. The frame storage interface is an extension of the frame retrieval interface. This follows the assumption that if you can write to a section of disk then you are implicitly able to read from that section as well, i.e. if you can write to the data store, you should be implicitly able to read from the data store as well. This service also provides the ability to delete records from the database. The Phoenix architecture defines two types of data stores: repositories and archives. Repositories are low-latency high-access data stores that should support higher data read and write rates. Archives are expected to be higher latency, low access data stores that may not be able to support high data rates but can store much more data than repositories. A possible implementation strategy would be to store recent data in a repository while aging data would be moved to an archive. This service may be implemented in such a way that it can be used as a wrapper for existing legacy data stores.

To see more specifics on this service and its methods, please consult with the documentation for the main stream repository service interface. StreamRepositoryService

Tier 4 - Administrative Services

These services are potentially optional services that provide backbone functionalities that may or may not be crucial to information management operations for a specific deployment of the Base Implementation services. For example, a set of Base Implementation services that are targeted to be deployed directly onto a camera pod (i.e. NCET services) may not require any service brokering or authorization capabilities and simply assume that all interactions are authorized and that all services are statically configured to know how to interact with each other. The set of Administrative Services includes:

- Authorization Service
- Service Brokering Service
- Session Management Service

Authorization Service

The Authorization Service (AS) is the security enforcement point for the Base Implementation. The implementation of this service is simplistic: it authorizes either everything or nothing. This design decision was made because a Base Implementation of a service should NOT be tied directly to any particular set of technologies.

Service Brokering Service

The Service Brokering Service (SBS) performs two basic functions: allows services to register themselves, and allows services to search (broker) for other services (stubs) for their own use. As most of the services are, the registration and evaluation of the services specifications are configurable. It's configurable in the attribute that can be used, as well as the way evaluation is done. The default set of attributes are listed below and it uses context query evaluator (contains a map of the attributes) to do brokering. When multiple fields are used for a search the SBS will assume an intersection operation upon the results (i.e. the multiple fields are conjoined using the 'AND' operation).

Configuration

Each service has two attributes relates to service brokering: 1) register with the SBS and 2) broker for services using the SBS. Set these values accordingly for each service.

Common Context Attributes

Below is a list of common context attributes that are in a Service Descriptor and can be queried over. They are only logically broken into two main groups: static (S) and dynamic (D). Static ones will be refined into attributes that either never/hardly change and ones that don't change that often, but not enough to be called dynamic (they will be called delineated with a *). This list of common attributes will be shared across all services. Then each service will have specific attributes only associated with that service. The dynamic ones below are currently just for review and discussion. They will not yet be part of the service descriptor or thus for brokering. The current list of attributes can be located in this class: mil.af.rl.phoenix.service.serviceregistration.ServiceDescriptorContext. Ones with a # below are the only ones that are currently being used in the system, and are common attributes (and in the ServiceDescriptorContext class).

Static

- **# HOST_ID**: String that is the name or ip of the machine (Ex: 128.132.60.71)
- **DOMAIN_ID**: String that is the name of the group of community or interest (COI).
- **# SERVICE_NAME**: String that is the name of this service (Ex: MainSubmissionService).
- **# SERVICE_TYPES:** *List* of Service Type Strings this service implements (Ex: SUBMISSION).
- **# SERVICE_CONTEXT_ID:** String that is a GUId (Ex: 6a2826ccf3cc9a557c3e479a4259f509).
- **# SERVICE_START_TIME**: DateTime in the format of example yyyy-MM-dd'T'HH:mm:ssZ (Ex: 2010-07-19T09:10:31-0400)
- *# **CONNECTOR_PROTOCOLS**: *List* of protocols. (Ex: RMI, PIC)
- # ***INFORMATION_TYPES**: *List* of current information types supported by this service (Ex: mil.n.ship)
- *# **CHANNELS**: *List* of current input channels for the service in the format of: channel_type:app_protocol_id:host_address:host_port:transport_protocol_id (Ex: serial:127.0.0.1:3149:tcp)

Dynamic

- NUM_CONNECTIONS: Int that is the number of connections to this service
- NUM_UNIQUE_CLIENT: Int that is the number of unique clients to this service
- CURRENT_RATE
- CONNECTIVITY_TIME
- UP_TIME
- NETWORK_AVG_AVAILABILITY
- TOTAL_RAM
- AVAILABLE_RAM
- NUM_CPUS
- CPU_SPEED
- CPU_PERCENT_USED
- SERVICE_AVG_AVAILABILITY
- OS_NAME_VER

Here is a list of some of the Phoenix Services and some of there possible service specific attributes:

Submission Service:

- NUM_CURRENT_PUBS
- AVG_PUBS_SEC
- INFO_BROKERS_URIS
- REPOSITORIES_URIS

Information Broker Service:

- CURRENT_PREDICATES_AND_TYPES
- RECENT_PAST_PREDICATES_AND_TYPES
- BROKERING_TECHNOLOGIES

Dissemination Service:

- DISSEMINATION_TECHNOLOGIES

Repository Service:

- REPOSITORY_TECHNOLOGY
- NUM_TYPES_SUPPORTED

- TOTAL_NUM_OBJECTS_STORED
- CURRENT_QUERIES_LIST
- NUM_PAST_QUERIES
- AVG_RESPONSE_TIME
- AVG_DESIRABILITY_INFO_TYPE
- AVAIL_DISK_SPACE

Session Management Service

The Session Management Service (SMS) is responsible for creating and maintaining sessions for actors. A session is intended to contain things like the information types that an actor is submitting, subscribing to, and querying over as well as any other items that an implementation may want to associate with a session and track. The SMS is the book keeper service that maintains the registry of all active and suspended sessions.

Support Packages

An alphabetical listing of the supporting packages for the Phoenix Base Implementation:

- Common Utilities
- Example Applications
- Example Config Applications
- Java Service Container
- Performance Applications

Common Utilities

This is a supporting project that has been developed by AFRL in coordination with the Base Implementation. It contains many utility functions that would otherwise have to be spread throughout the Base Implementation. The Common Utilities project consists of nine component packages:

- Benchmark
- Buffering
- Common
- Database
- Encoding
- GUI
- Logging

- VO
- XML

Benchmark

The benchmark package provides classes that perform some level of benchmarking or metrics gathering, like timed events for average rate calculations.

Buffering

TimerBasedBuffer

This class is used to provide a common buffering engine and interface. The buffer's design utilizes Java generics to provide an easily extensible utility class for buffering Java objects. This class works off the Template Design Pattern the class itself being abstract with the *doSpendBuffer*(List<E> spendBuffer) method. Only sub-classes know what to do with the list of objects and handle them appropriately (i.e: spend them). The properties of this class (the way that is works), is governed by some default attributes, that can be set using a property file, see TimerBasedBuffer.properties.

TimerBasedBuffer.properties

This file governs how buffering will work. The file location is relative to the directory of where the application has started from. If it does not find it in that root directory, it will search its parents path, until it's at the root of the OS.

Here are the list of properties that can be set. NOTE: More about these are the appropiate values can be seen in the javadocs for this class.

- **maxBufferSize -** The maximum number of elements the buffer can contain.
- **spendSize -** The Maximum number of elements that will be "spent" from the buffer every Spend Interval. This must be > 0 and <= Max Buffer Size.
- **spendEagerness -** The amount of time in milliseconds the spend thread will wait before attempting to spend Max Spend Size items from the buffer.
- **sleepOnMaxBufferSizeTime -** If the buffer is already at max size when add() is called, the call to buffer will be blocked for the specified number of milliseconds. If after that time the buffer is still full the call will be blocked again for the same amount of time. This blocking behavior continues until there is room in the buffer for the new item.
- **sleepOnAddToBufferTime -** Blocks on all calls to add() for the specified number of milliseconds (simple throttling).
- **decimateByEvery -** Specifies that only every n calls to buffer should actually result in an item being added to the buffer. For example if you specified a value of 5. Only every 5th call to add() would result in the items passed in being buffered.

- **maxBufferSpendSizeLogCount -** When enabled (!= -1) the Buffer has reached with-in a spend size of its Max Size, log it as instrumentation data.
- **maxBufferFull -** When enabled (== 1) logs when the buffer becomes full and how long it took it to become full. NOTE: maxBufferSpendSizeLogCount must be enabled for this to be enabled.
- **sleepOnRecoverableSpendFailure -** This is the time in milliseconds to wait after a failure has occurred before it tries to spend again [call spendBuffer()]
- **showLogGUI -** Show a buffering GUI in which it charts the size of the buffer against its max size over time.

Each of the properties above should be prefaced with either the name of the concrete Buffer Class name or the name you gave the buffer upon instantiation (constructor call) followed by a period '.' ending with that property name. So, by default if the class name is SubmitTimerBasedBuffer.java, then values in the property file could look like:

```
SubmitTimerBasedBuffer.maxBufferSize = 5000
SubmitTimerBasedBuffer.spendSize =  100
SubmitTimerBasedBuffer.spendEagerness = 10
SubmitTimerBasedBuffer.sleepOnMaxBufferSizeTime = 10
SubmitTimerBasedBuffer.sleepOnAddToBufferTime = 0
SubmitTimerBasedBuffer.decimateByEvery = -1
SubmitTimerBasedBuffer.maxBufferSpendSizeLogCount = 1
SubmitTimerBasedBuffer.maxBufferFull = 1
SubmitTimerBasedBuffer.sleepOnRecoverableSpendFailure = 2000
SubmitTimerBasedBuffer.showLogGUI=0
```

But, you could have 2 or more instance of the buffer in your code, and want to have different settings for each, so you can have a 'meaningful buffer name', like: InputSubmitTimerBasedBuffer and 'OutputSubmitTimerBasedBuffer'. So, you would have 2 sets of property values in the file, each having its own set of values.

Buffer Values

```
maxBufferSize:
   0 - This will make buffering NOT really buffer at all (its a synchronous call). It will act as a pass-thru.
       Meaning, it will add to the buffer and we alone will spend immediately (no threads),
       thus like a real blocking synchronous call.
   > 0 - Size of the buffer
   < 0 - [ERROR]

spendSize:
   > 0 - Size to spend at a given time
  <= 0 -  [ERROR]
       NOTE: spendSize > maxBufferSize - is also an [ERROR]

spendEagerness:
   > 0 - Time in mills to call spend on a timer
   = 0 - [WARNING] - Will continue to spend immediately. Can do, but probably not wanted
   < 0 - [ERROR]

sleepOnMaxBufferSizeTime:
   > 0 - Time in mills to sleep and wait for some of the buffer to be spent
```

<= 0 - Will not sleep, but will remove one, allowing the next to be added (revolving buffer)

sleepOnAddToBufferTime:
> 0 - Time in mills to sleep on all adds to buffer (simple throttling)
<= 0 - Will ignore and NOT sleep on all adds

decimateByEvery:
-1 - Do NOT decimate {Always add to the buffer, like normal}
0 - Always decimate {Decimate every one, never add to the buffer (like stopping a pub'bing sensor)
}
>= 1 - We WILL decimate (keep) every n'th one, all the rest are dropped on the floor.

NOTE: 1 means decimate by 1 (like -1), so we keep every one, thus always added to the buffer, never dropped

Common

This package contains raw utility functions that can be applied to many Java objects. This section will break the package down by implementation class and provide a brief overview of the functions that each class offers.

EmptyUtils

This class provides methods for checking if a given object is empty. These methods all perform null checks on the given value first and only if they are not null will the actual check for emptiness be performed. These methods allow developers using Common-Utilities to remove many null checks from their own code, thereby cleaning it up a bit.

FileUtils

This class provides some common file utilities including reading and writing files, property files, finding/locating files, and deleting files.

MultipleValueMap

This class is a synchronized map implementation that allows for multiple values for any given key.

ObjectAnalyzer

This class provides methods for converting any Object into a human readable string and for comparing any two given objects to each other.

PropertiesUtils

This class provides methods for converting java.util.Properties instances to strings and vice versa.

Serializer

This class provides methods for generic serialization and de-serialization of Java objects to and from byte arrays, respectively.

SortedProperties

This is an extension of the java.util.Properties class that returns the key names in alphabetical order through an overloaded 'keys()' function.

SpringUtils

This class provides a method for locating a Spring configuration file given a file name, path, and bean name. This function accepts wildcarded (Example: **/src/text/resources/phoenix_beans.xml) or absolute paths. The file being referenced must be on the Java classpath in order to be found by this function. Once found, the file must contain the given bean name or an exception is thrown.

StreamUtils

This class provides some general streams utility functions like copying streams or reading them into a String variable.

StringUtils

This class provides a method for discovering if a StringBuffer ends with a given suffix.

Utils

This class provides some general utility methods including converting hexadecimal values to byte arrays, converting strings to enumeration values, and retrieving the class name from a fully qualified class name.

Database

This package contains classes whose methods provide some common functions consistent across many Java Database Connection (JDBC) clients.

Encoding

This package contains classes and methods for encoding and decoding objects and strings to and from Base 64 byte arrays, respectively. It also contains a class for generating random GUIDs based on the Java MD5 has generator.

GUI

This package contains methods for constructing form- or grid-style layouts with SpringLayout.

Logging

This package defines a specific Apache Log4J logger class named OIMLogger, its specific logging levels, factory, and file appender. It also contains a specific implementation of a Log4J network logger service.

The log4j service is run to collect logging events (messages) that are sent over a specific port. We use this to collect information from the pub/sub Performance applications and JSC in order to place all metric log statements (and all other level), so that they are located in a single log file for parsing.

The service is started/run from the performance apps, since that is where it will be mostly used. To start it, run this: \base-implementation\performance\log.bat

Configure Service

This script runs the ant build script calling the 'log' ant target. That target runs the mil.af.rl.phoenix.util.logging.net.OIMSocketServer class. This is really just a wrapper class that extends the org.apache.log4j.net.SocketServer class. We have this class setup, just in case in the future we want to control the log events (messages) that come in.

To change the port that the log4j service runs on, change the log.service.port ant variable in the file or override it via ant properties. The \base-implementation\performance\src\main\resources\log4j-service.xml configures how this service will handle log events/messages it received on a socket.

Configure Client

To change the host or port used for the performance apps, edit it's \base-implementation\performance\src\main\resources\log4j.xml file. If you want to add this to an existing file, just copy/paste the below in:

```
<appender name="SOCKET" class="mil.af.rl.phoenix.util.logging.net.OIMSocketAppender">
        <param name="RemoteHost" value="127.0.0.1" />
        <param name="Port" value="8887" />
</appender>
```

The OIMSocketAppender just extends log4js' SocketAppender class. We could at somepoint chance the functionality here, and now we are setup for that.

VO

This package contains some basic utility methods for value objects.

XML

This package contains some basic XML functions and a simple XML validator that uses the ISO RELAX verifier.

Example Applications

The Example Applications are example Java applications that show the following Phoenix concepts:

1. **PublishClient.java** (simplePub.bat)- shows how to submit information to the Phoenix services via the Submission Service Stub.

2. **SubscribeClient.java** (simpleSub.bat) - shows how to subscribe for information from the Phoenix services via the Information Brokering Service Stub.

3. **QueryClient.java** (simpleQuery.bat) - shows how to query for information from the Phoenix services via the Query Service Stub.

4. **TypeClient.java** (simpleTypes.bat) - shows how to add a type to the Phoenix via the Information Type Management Service Stub.

Pre-Requisites

- There are Batch (.bat) and Shell (.sh) scripts available in the \base-implementation\exampleapps directory. These both call Apache Ant to run the clients. The Ant script (build.xml) is located in the same directory. Thus, Apache Ant is required to run these example applications (see: http://ant.apache.org) for more information). NOTE: These applications are compiled by using Maven when in the main/root level pom.xml file in \base-implementation\ is executed.

- In order to run these clients, the needed Phoenix services should be configured and running. The Java Service Container (JSC), has a pre-configured set a services and workflow built in, and thus recommended to use while running these example applications.

- The Type Client needs to be run before the PublishClient and QueryClient, if the persist option is 1 and the JSC default configuration is changed not to create and store the type: mil.n.ship.

- The ant script defaults to using the exampleapps.properties file, but this can be changed via command line. Just run this command: >ant -buildfile=buildSimple.xml -DpropertiesFile=my.properties sub

The Clients

The clients are intended to show how to use the services and perform those basic operations. They can be copied and used appropriately. Some of them are parameterized somewhat, were others are more hard-coded. Also, some of the clients have some metrics information. The clients have some basic metric information related for publish, subscribe, and query times. The clients parameters are set in the exampleapps.properties file and are read in using the Ant build.xml script, and passed into java. The script and property files are located here: \base-implementation\exampleapps.

1. **PublishClient:** Parameters are: number of Information Objects to publish, number of types to use, and whether to persists the information to a database (1 or 0)

2. **SubscribeClient:** Parameters are: number of Information Objects to publish and number of types to use

3. **QueryClient:** Parameter is: query string

4. **TypeClient:** No parameters

In general the flow of each client is the same, setup the RMI stubs, parse any incoming parameters, setup any channels and/or callbacks, invoke the control calls on those stubs , and then it should start publishing/receiving information. The TypeClient only invokes control methods. It first checks to see if the type is already created, and if not, it creates it.

Example Config Applications

The Example Config Applications are example Java applications that show the following Phoenix concepts:

1. **PublishConfigClient.java** (pub.bat)- shows how to submit information to the Phoenix services via the Submission Service Stub.
2. **SubscribeConfigClient.java** (sub.bat) - shows how to subscribe for information from the Phoenix services via the Information Brokering Service Stub.
3. **QueryConfigClient.java** (query.bat) - shows how to query for information from the Phoenix services via the Query Service Stub.
4. **TypeConfigClient.java** (types.bat) - shows how to add a type to the Phoenix via the Information Type Management Service Stub.

Pre-Requisites

- There are Batch (.bat) and Shell (.sh) scripts available in the \base-implementation\exampleapps directory. These both call Apache Ant to run the clients. The Ant script (build.xml) is located in the same directory. Thus, Apache Ant is required to run these example applications (see: http://ant.apache.org) for more information). NOTE: These applications are compiled by using Maven when in the main/root level pom.xml file in \base-implementation\ is executed.
- In order to run these clients, the needed Phoenix services should be configured and running. The Java Service Container (JSC), has a pre-configured set a services and workflow built in, and thus recommended to use while running these example applications.
- The Config Type Client needs to be run before the PublishConfigClient and QueryConfigClient, if the persist option is 1 and the JSC default configuration is changed not to create and store the type: mil.n.ship.
- The ant script defaults to using the configapps.properties file, but this can be changed via command line. Just run this command: >ant -DpropertiesFile=my.properties sub

The Clients

The clients are intended to show how to use the services and perform those basic operations. They can be copied and used appropriately. The client's parameters are set in the configapps.properties file and are read in using the Ant build.xml script, and passed into java. The script and property files are located here: \base-implementation\exampleapps.

In general the flow of each client is the same, setup the RMI stubs, setup any channels and/or callbacks, invoke the control calls on those stubs, and then it should start publishing/receiving information. The TypeClient only invokes control methods. It first checks to see if the type is already created, and if not, it creates it.

These config clients are governed by a Spring Framework (from http://www.springsource.org) config file. This file contains all the necessary information for each of the 4 clients. It contains the client configurations for publishing, subscribing, querying, and creating types. The contents of each client configuration is governed by that clients java methods (javadoc), as the Spring Framework uses Inversion of Control (IoC). Each bean in that file represents a client that is being created and configured.

Spring Framework Links:

- http://www.roseindia.net/spring
- http://courses.coreservlets.com/Course-Materials/spring.html
- http://static.springsource.org/spring/docs/2.5.x/spring-reference.pdf OR
- http://static.springsource.org/spring/docs/2.5.x/reference/index.html

Java Service Container (JSC)

The Java Service Container (JSC) is a very lightweight Plain Old Java Object (POJO) service container. The JSC holds a list of Service Managers, and each of these contains a service instance and a connector manager for that specific service. Each Service Manager runs as a separate and distinct thread of execution, able to be managed and prioritized like any other Java thread.

The JSC is not intended to replace an application server. Instead it provides a convenient and simple framework to configure, start, stop and lightly manage services in a single JVM. In fact, one or more instances of the JSC could live within an application server. The JSC takes a list of pre-configured contexts and services. The JSC relies on an external component to configure the services and other entities that it maintains. The JSC provides a common, simplistic interface for managing and retrieving service instances by service name or type.

Building the JSC

The JSC runtime libraries are gathered and setup during the execution of the Maven build through the use of the *copy-dependencies* plugin. This tells Maven to look at all the dependencies (jar's, dll's, so's) of the JSC and copy them into the default dependency directory: \base-implementation\javaservicecontainer\target\dependency. The Ant build script that runs the JSC is configured to use this directory as its home directory. Other required files such as the Spring and Log4J configurations are located in the /target/classes directory.

Configuring the JSC

Spring Loaded JSC

Now, loading the JSC with pre-configured services might seem cumbersome and trite assuming you had to create a JSC loader that had hard-coded values for the contexts and service settings. But, enter stage left to save the day, the Spring Framework (from http://www.springsource.org) and the SpringJSCLoader provide a dynamically configurable setup for the JSC and its services.

The SpringJCSLoader class itself is pretty short and sweet. It takes an optional file name that defaults to 'phoenix.service.jsc_beans.xml'. This Spring configuration file is loaded into the Spring Framework and returns a fully configured and ready to use JSC. When used as the main class for starting/loading JSC, it even starts all the services (meaning it looks over all the services and calls it's start() method).

Spring Configuration File ('phoenix.service.jsc_beans.xml')

Spring works off the bean specification of getters/setters (mostly setters). The SpringJscLoader class takes a file that corresponds to the Java API of the JavaServiceContainer class. An example JCS Spring configuration file is located here: phoenix.service.jsc_beans.xml.

A quick glance at this file reveals a myriad of complicated XML nodes. **Do Not Panic** There is a method and an organization to the madness!

First and foremost look at just the 'bean' node with the ID of "phoenixJavaServiceContainer". This is the actual JSC Java class that will be configured with Service Manager instances. Next notice that this node contains one property named "services" that is itself a list of Service Managers. Each of these Service Managers are linked to a service 'bean' node in this same configuration file. Each Service Manager node also defines zero to *n* connectors for its corresponding service. Thread priority for services is also set within each service's Service Manager configuration node. It is also important to note here that the order that the Service Managers are defined in directly corresponds to the order that the JSC will start the individual services in.

Listed immediately following the JSC node will be the set of individual service instance configuration nodes. In the provided example configuration file, all service instance ID's end with the moniker 'Service', but this is not required. Each service configuration node contains the configuration node for its specific Service Context instance, the list of configuration nodes for the serivce's available input channels, and any other settings (stubs to connect to other services with, etc.) pertinent to the service.

Pre-Configured Information Type for Example Applications

Both the Repository and Information Type Management Service's are pre-configured for the type: mil.n.ship. This also the full suite of Example Applications to be run without first running the TypeClient Example Application.

Depends Service Order (dependsOnServiceNames)

The Service Manager has get/set dependsOnServiceNames methods that takes a list of strings. These lists of strings are the names of the other services that the given service (e.g. its service manager) are dependent upon starting first. So, list the service names of those services that needs to be started first, before the given service can be started. Normally, this would be those services that a service has stubs setup for, as well as channels configured to them. The JSC will figure out the correct start order that the services should be started in.

Running the JSC

Located in the root directory: \base-implementation\javaservicecontainer, there are two scripts (jsc.bat and jsc.sh) that will start the JSC on Windows and Unix respectively. They both call an Ant script that calls Java to run the main method in the SpringJscLoader class. Or you can use a command prompt to navigate to the home directory and enter '>ant' to run the default Ant

target 'run.Jsc'. The Ant variable 'jsc.toload.file' defaults to 'phoenix.service.jsc_beans.xml' tells the Spring what configuration to load into the JSC. An example of changing the file name that is used for loading the JSC via the SpringJscLoader class is:

```
ant -Djsc.toload.file=my_jsc_beans.xml
```

In order to run a self sustained version (one that won't be changed from one build to the next) of the JSC, you will probably want to copy the directory structure into another location, so that the jars and files do not get replaced by executing a Maven 'clean' command. This will ensure that you are guaranteed the same libraries between JSC runs. You will probably need to change the Ant script if you still want to use Ant to kick-off the JSC.

Spring Framework Links:

- http://www.roseindia.net/spring
- http://courses.coreservlets.com/Course-Materials/spring.html
- http://static.springsource.org/spring/docs/2.5.x/spring-reference.pdf OR
- http://static.springsource.org/spring/docs/2.5.x/reference/index.html

Performance Applications

There are various performance applications, but we will just focus on the two main clients.

1. **PubInfoSubmissionPerfClient.java** (pub.bat)- Submit information to the Phoenix services via the Submission Service Stub. It also calculates metrics as the publisher preceives the information being published.
2. **SubInfoInfoBrokerServicePerfClient.java** (sub.bat) - Subscribe for information from the Phoenix services via the Information Brokering Service Stub. This client calculates

Pre-Requisites

- There are Batch (.bat) and soon to be Shell (.sh) scripts available in the \base-implementation\performance directory. These both call Apache Ant to run the clients. The Ant script (build.xml) is located in the same directory. Thus, Apache Ant is required to run these example applications (see: http://ant.apache.org) for more information). NOTE: These applications are compiled by using Maven when in the main/root level pom.xml file in \base-implementation\ is executed.
- In order to run these clients, the needed Phoenix services should be configured and running. The Java Service Container (JSC), has a pre-configured set a services and workflow built in, and thus recommended to use while running these example applications.
- The ant script defaults to using the configperfapps.properties file, but this can be changed via command line. Just run this command: ant -DpropertiesFile=my.properties sub

The Clients

These performance clients are governed by a Spring Framework (from http://www.springsource.org) config file. This file contains all the necessary information for all of the clients. It contains the client configurations for publishing, subscribing, querying, and creating types. This contents of each client configuration is governed by that clients java methods (javadoc), as the Spring Framework uses Inversion of Control (IoC). Each bean in that file represents a client that is being created and configured. The clients by defaults are governed by this spring file: \base-implementation\performance\src\main\resources\phoenix.perf_client_beans.xml

Spring Framework Links:

- http://www.roseindia.net/spring
- http://courses.coreservlets.com/Course-Materials/spring.html
- http://static.springsource.org/spring/docs/2.5.x/spring-reference.pdf OR
- http://static.springsource.org/spring/docs/2.5.x/reference/index.html

Third Party Libraries

Third party libraries offer abilities that expand and enable the Base Implementation. This section will enumerate all third party libraries utilized by the Base Implementation and explain what each does and why it was chosen.

- Berkeley DB XML
- Mockets
- XPP3
- XStream

Berkeley DB XML

The Berkeley Database has two exclusive components: Berkeley DB and Berkeley DB XML. Berkeley DB (BDB) is a key-value pair database that is loosely comparable to a Java Hashtable construct while Berkeley DB XML (BDBX) is a Native XML Database (NXD) designed to store raw XML documents. Retrieval of values from BDB consists of supplying the database with a key to retrieve the corresponding value for while retrieval of records from BDBX requires a valid XQuery statement.

Berkeley currently supports a host of programming languages via the Simplified Wrapper and Interface Generator (SWIG) and native Berkeley Application Programming Interfaces (API). This current set includes: C++, Java, Perl, PHP, Python, and Tcl. C#/.NET and Ruby are not supported internally but external solutions are available.

Berkeley DB XML 2.5 Release Overview [3]

Release 2.5 is primarily a feature release with a small number of useful features:

- Automatic indexing of leaf elements and attributes

- Whole Document container compression with optional user-defined compression in C++ and Java
- Improvements in node storage containers that reduce total size of containers
- User-defined external XQuery extension functions in C++, Java and Python
- XQuery debug API in C++, Java and Python
- Improvements in the XmlResults class enabling better offline results handling

Berkeley executes as an embedded database within a hosting application's memory space. The Berkeley databases, as supplied by Oracle, do not function as standalone services in the traditional database manner. Rather, in order to execute, Berkeley requires that several Dynamic Link Library (DLL) files be available somewhere on the parent application's PATH and that Berkeley's resources be managed by your custom application. Berkeley databases, both the BDB and BDBX, function as in-memory databases with an option to write data to disk in specially formatted files. The complete list of Berkeley DB XML libraries is as follows:

Library Name	Type	Version
Berkeley DB JE	JAR	2.5.13
Berkeley DBXML JE	JAR	2.5.13
msvcp	DLL	7.1
msvcr	DLL	7.1
xerces-c	DLL	3.0
xqilla	DLL	2.2
zlib	DLL	1.0

For Linux builds, replace all DLLs with the following Shared Objects (SO) libraries.

Library Name	Type	Version
libdb-4.6	SO	4.6
libdb_cxx-4.6	SO	4.6
libdb_java-4.6	SO	4.6
libdbxml-2.4	SO	2.4
libdbxml_java-2.4	SO	2.4

libxerces-c	SO	2.8
libxqilla	SO	4.0.3

Mockets

Mockets (for “mobile sockets”) is a comprehensive communications library for applications. “Mockets is a comprehensive communications library designed to address challenges specific to mobile ad-hoc networks. Mockets have been implemented at the application-level to simplify deployment and portability. Both stream-oriented and message-oriented abstractions are supported, with the message-oriented service providing multiple classes of service (reliable, unreliable, sequenced, unsequenced), message tagging and replacement, and prioritization. Mockets also interfaces with a policy management infrastructure to support bandwidth limitation. Finally, mockets supports transparent migration of communication endpoints across hosts without the need to terminate and reestablish connections. Mockets provides similar semantics to TCP but performs better than TCP on adhoc networks.” [1]

XPP3

The XML Pull Parser (XPP) is a streaming XML pull parser quickly processes XML inputs. For more information about XPP3 refer to the web-site: http://www.extreme.indiana.edu/xgws/xsoap/xpp

Library Name	Type	Version
xpp3	JAR	1.1.4c
xpp3_xpath	JAR	1.1.4c

XStream

XStream is a library that serializes Java Objects to XML and restores them. For more information about XStream, refer to its web-site: http://xstream.codehaus.org

Library Name	Type	Version
xstream	JAR	1.3.1

Requirements

1. Implement each and every component and service interface at a basic level.

- Implement all services as Plain Old Java Objects (POJO).

2. Implement information definition, processing, & delivery that supports both attribute-value pair and XML metadata.
    - Information Type Management
        1. Implement information type definitions using attribute-value pair based metadata.
        2. Implement information type definitions using XML based metadata.
    - Information Brokering
        1. Implement information brokering that supports simple attribute-value pair operations.
        2. Implement information brokering that supports the full range of XPath 1.0 syntax and operations.
3. Implement service definition and processing that supports XML service descriptions.
    - Service Brokering
        1. Implement support for describing services using XML.
        2. Implement service brokering that supports the full range of XPath 1.0 syntax and operations.
4. Implement a simple POJO service container.
    1. POJO container shall be able to start and stop services in a specified order.

Provide simple exemplary edge actor classes that show how to perform basic information management operations using Phoenix constructs and services.

- Information Publish & Subscribe
- Information Persistence & Retrieval
- Information Type Definition & Registration

Testing

The Phoenix Base Implementation shall be thoroughly unit, integration, and performance tested using a variety of commercially available and home-grown testing products and methodologies.

Unit Testing

Unit testing of all developed project components and services will be accomplished through the use of the JUnit testing framework. JUnit was chosen because it is becoming the de facto standard within the Java development world for unit testing and it has well-supported plug-ins

for both the Eclipse IDE and the Maven build environment. The current version of JUnit used for unit level testing is 4.4.

Unit test coverage reports will be generated by the Cobertura test analysis tool. This tool plugs into the Maven build environment and will be configured to run whenever the web-site is built for a Maven project module. Cobertura offers insight as to what executable lines of code have been tested, how many times they have been executed, and which conditions of a conditional branch have been satisfied by the tests being run. The reports generated divide executable code coverage numbers and conditional branch coverage numbers. The current version of Cobertura Maven plug-in being used is 2.2.

A sample Cobertura report is shown below.

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	47	83%	969/1168	87%	218/250	1.989
mil.af.rl.phoenix.channel	7	99%	160/162	100%	36/36	1.029
mil.af.rl.phoenix.channel.control.connectors	1	100%	2/2	N/A	N/A	1
mil.af.rl.phoenix.channel.control.stubs	1	100%	2/2	N/A	N/A	1
mil.af.rl.phoenix.channel.data	5	96%	72/75	83%	15/18	1
mil.af.rl.phoenix.channel.data.information	3	97%	38/39	100%	10/10	0
mil.af.rl.phoenix.channel.data.information.block	2	100%	25/25	100%	8/8	2
mil.af.rl.phoenix.channel.data.information.mocket	3	88%	59/67	100%	16/16	3
mil.af.rl.phoenix.channel.data.information.stream	2	100%	25/25	100%	8/8	2
mil.af.rl.phoenix.channel.data.transport	6	78%	66/85	88%	7/8	1
mil.af.rl.phoenix.channel.data.transport.mockets	5	62%	156/250	71%	30/42	5.714
mil.af.rl.phoenix.channel.data.transport.tcp	3	83%	79/94	83%	20/24	3.308
mil.af.rl.phoenix.channel.data.transport.udp	2	79%	78/99	100%	20/20	0
mil.af.rl.phoenix.channel.service	3	90%	114/126	70%	21/30	0
mil.af.rl.phoenix.channel.util	4	80%	44/111	90%	27/30	3.667

Report generated by Cobertura 1.9 on 4/21/09 8:00 AM.

Integration Testing

Integration testing of the Base Implementation services is carried out through the development of specific JUnit test cases that are contained in their own project module insightfully named: "integrationtest". These integration JUnit tests do NOT use the Spring configuration files to configure and interrelate the services. Service configuration and relationships are created at JUnit test setup time and are hard coded into the individual test classes. This was purposefully done to remove the Spring configuration and setup as an extra variable when testing the relationships between Base Implementation services.

Existing Integration Tests

The current set of integration tests is as follows:

- Broker and Disseminate
- Broker and Notify
- Submit and Validate
- Submit, Broker, and Disseminate
- Submit, Broker, and Notify

- Submit, Broker and Store, and Disseminate
- Submit and Store
- Submit, Store, Query, and Disseminate

Broker and Disseminate

This integration test verifies the relationship between the Information Brokering Service and the Dissemination Service when registering subscriptions for information and then brokering submitted information. This test focuses specifically on the link between the IBS and the DS by removing the Submission Service from the playing field. This test is limited to a single IBS and a single DS.

Broker and Notify

A test that verifies the relationship between the Information Brokering Service and the Event Notification Service by testing the operations of the ENS. During this test an event consumer registers for an event notification and then listens for a corresponding event. The event consumer in this test is a simple thread, controlled by the test class, that opens and reads from an Event Input Channel. This test adds the Submission Service to the Broker and Notify test environment to check that information submission operations do not affect the outcome of an information brokering operation that results in an event notification operation.

Submit and Validate

A test that verifies the relationship between the Submission Service and the Information Type Management Service by testing the validation of submitted XML information instances. This test uses one instance of each service. No actual checks are included in the test code because no exceptions are thrown back to the producer (in this case the test code is the producer). An exception reported via Log4J is expected to be generated by this test.

Submit, Broker, and Disseminate

This integration test verifies the basic publish and subscribe functionality provided by configuring a Submission Service, Information Brokering Service, and Dissemination Service into a functional chain. This test focuses on the submission and reception of information and fails if anything goes wrong at any point in the operational or functional chain of events that are inherent in publish and subscribe operations. This test is limited to a single SS, IBS, and DS. This test does not include a Repository Service, the SS is configured to only forward submitted information to the single available IBS.

Submit, Broker, and Notify

A test that verifies the relationships between the Submission Service, Information Brokering Service, and Event Notification Service during information brokering operations that result in an event notification operation. During this test an event consumer registers for an event notification and then listens for a corresponding event. The event consumer in this test is a simple thread, controlled by the test class, that opens and reads from an Event Input Channel.

Submit, Broker and Store, and Disseminate

A test that verifies the relationships between the Submission Service and a single Information Brokering Service and a single Repository Service. This test would verify that a single instance of information can be sent to two different services over two different information channels for two different operations. This test also includes the information dissemination operation since we want to fully test the integration of publish and subscribe with the storage operation. The consumer in this test is a simple thread, controlled by the test class, that opens and reads from an Information Input Channel.

Submit and Store

This test verifies the basic relationship between a Submission Service and a Repository Service. This test checks to make sure that all submitted information instances have been stored in the correct locations for possible later retrieval. This test utilizes the Berkeley DB XML repository as the data store for the RS. This test is limited to a single SS and a single RS. This test does not include an Information Brokering Service, the SS is configured to only forward submitted information to the single available RS.

Submit, Store, Query, and Disseminate

This test verifies the basic relationship between a single Repository Service and a single Query Service. This test checks to make sure that the submitted information instances have been stored, the correct number of instances match the query issued to the QS, and the correct information instances are returned to the consumer. This test also utilizes a single Submission Service for information submission and forwarding to the RS as well as a single Dissemination Service for distribution of the query results to the consumer. The consumer in this test is a simple thread, controlled by the test class, that opens and reads from an Information Input Channel.

Performance Testing

Performance testing was performed using standalone clients and a single Java Service Container hosting a standard set of Phoenix Services. A suite of three machines served as the testbed for the performance tests. The network for the test was a 100/1000 Gigabit link network, with all machines having supporting network cards installed as their primary network connection. Two machines were managed by Windows XP while one was managed by Ubuntu 10.4, thus proving the cross-OS compatibility of the Base Implemenation's communications mechanisms.

The clients included simple information producer, consumer, inquisitor, type management, session management and event firing and reception. Each client was developed in Java using the Phoenix Base Implementation. The clients are executed by an Ant script that also contains the configuration settings for each client. These settings are passed into the client as command line arguments.

The standard set of Phoenix services deployed for these tests includes one instance each of the: Authorization Service, Dissemination Service, Event Notification Service, Information Brokering Service, Information Type Management Service, Query Service, Repository Service, Service Brokering Service, Session Management Service, and Submission Service.

All performance tests were run ten times and the results averaged to produce the final reported metrics. All tests performed utilized the network, with no co-hosting of clients either with each other or with the JSC.

The throughput and latency of the Submission, Information Brokering, and Dissemination Services were tested by submitting a set of information isntances of a known size. The throughput of the services were measured in objects per second while the latency was measured in seconds per object. These tests were accomplished having the information consumer client configure the service at runtime to forward processed information to it. For example, to test the Dissemination Service a static configuration for the consumer client was part of the information instance that was submitted to the service by the information producer client.

Additional performance and stress testing was performed using the High Performance Computing (HPC) EmuLab cluster. All test results were compiled using spreadsheets and formed the basis for an accompanying performance testing technical report.

Reference

[1] Arguedas, M., Breedy, M., Carvalho, M., Suri, N., Tortonesi, M., Winkler, R., “Mockets: A Comprehensive Application-Level Communications Library”, Defense Technical Information Center, 2005.

Reference

Documents

1. Lipa, Brian. "Berkeley Overview". *AFRL In-House Research (Non-Published)*, May, 2009.
2. Sun MicroSystems, Inc. "Java Code Conventions". *http://java.sun.com/docs/codeconv/CodeConventions.pdf*, 12 Sepetember 1997.
3. Oracle. "2.5 Release Overview". *http://www.oracle.com/technology/documentation/berkeley-db/xml/ref_xml/changelog/2.5.html*, September, 2009.

Terms and Acronyms

The table below gives a brief description of important terms and acronyms used in this document. For definition of Phoenix Architecture terms and acronyms refer to its specification.

Term / Acronym	Meaning
AFRL	Air Force Research Laboratory.
API	Application Programming Interface.
AS	Authorization Service.
ASD	Attribute Schema Document.
BDB	Berkeley Database.
BDBX	Berkeley Database XML.

CRS	Client Runtime Service.
DLL	Dynamic Link Library.
DoD	Department of Defense.
DOM	Document Object Model.
DS	Dissemination Service.
DTD	Document Type Definition.
ENS	Event Notification Service.
ESB	Enterprise Service Bus.
FMS	Filter Management Service.
HTTP	Hypertext Transfer Protocol.
IA	Information Assurance.
IBS	Information Brokering Service.
IDS	Information Discovery Service.
IM	Information Management.
ITMS	Information Type Management Service.
JAR	Java Archive.
JDK	Java Developer's Kit.
JSC	Java Service Container.
JVM	Java Virtual Machine.
LFS	Local File System.
MSV	Multi-Schema Validator.
NFS	Network File Share.
NXD	Native XML Database.
PEP	Policy Enforcement Point.

PHP	PHP: Hypertext Preprocessor.
PMD	No official definition, several unofficial including *Programming Mistake Detector*.
POP	Point of Presence.
POJO	Plain Old Java Object.
QS	Query Service.
RDBMS	Relational Database Management System.
RI	Reference Implementation.
RMI	Remote Method Invocation.
RS	Repository Service.
SDK	Software Development Kit.
SO	Shared Object.
SOA	Service Oriented Architecture.
SBS	Service Brokering Service.
SMS	Session Management Service.
SS	Submission Service.
SUS	Subscription Service.
SWIG	Simplified Wrapper and Interface Generator.
TCP	Transport Control Protocol.
UDP	User Datagram Protocol.
UI	User Interface.
XBS	Stream Brokering Service.
XDS	Stream Discovery Service.
XML	Extensible Markup Language.
XPP	XML Pull Parser.

XRS	Stream Repository Service.
XSD	XML Schema Document.
XSLT	Extensible Stylesheet Language Transformations.

Releases

This section lists all releases made of the Phoenix Base Implementation including short descriptions of why each release was created. The row colors indicate whether or not the release process result in a useable version of the software. Red indicates a release that failed to meet the Quality Assurance standards of the group and therefore was not delivered to any interested parties. Green indicates a successful release that may have been shipped to one or more external groups.

Version	Release Description
1.0.0	Internal release used to setup and configure the release process.
1.0.1	Internal release used to trouble shoot the release process.
1.0.2	Internal release used to trouble shoot the release process.
1.0.3	Initial release created for baseline purposes. No functionally complete modules included.
1.0.4	Internal release prior to 1JUN delivery date. Discarded due to inconsistencies.
1.0.5	Internal release prior to 1JUN delivery date. Discarded due to inconsistencies.
1.0.6	Release created for 18JUN delivery date. Included functionally complete modules: Channel, Core, Information, Information Type (just contexts, not ITMS)
1.0.7	Internal release prior to 1AUG delivery date. Discarded due to inconsistencies.
1.0.8	Internal release prior to 18AUG delivery date. Discarded due to inconsistencies.
1.0.9	Release created for 18AUG delivery date. Added functionally complete modules: Dissemination, Event, Event Notification, Information Brokering, Information Type, In-Memory Platform, Expression, Submission
1.1.0	Internal release prior to 1OCT delivery date. Discarded due to Typed File Repository JUnit test class causing a hang during the build process.
1.1.1	Internal release prior to 1OCT delivery date. Bug fix for threading issues in Typed File

	Repository JUnit test. Discarded due to <100% dependency convergence.
1.1.2	Release created for 1OCT delivery of Base Implementation code base. Fixed dependencies to achieve 100% convergence. Includes Base context update attribute callback patch from BBN. Added functionally complete modules: Example Apps, Java Service Container (formerly In-Memory Platform), Integration Test, Query, Repository (except move to archive methods), Session, Session Management
1.1.3	Internal release created for the inclusion of the asynchronous channels. No new modules added to project. This release was found to have intermittent build issues within the integration tests module and was thus quickly replaced with build 1.1.4.
1.1.4	A finalized, official release of the Base Implementation suitable for delivery to external contractors. No new modules were added to this project.
1.1.5	A finalized, official release of the Base Implementation suitable for delivery to external contractors. Added functionally complete modules: Frame, Connection, Stream Brokering, and Phoenix Invocation Control (PIC). This release was officially tagged: "Lead" (as in the 82nd element on the periodic table)
1.1.6	A finalized, official release of the Base Implementation suitable for delivery to external contractors. In actuality, this release was created solely for the purpose of base-lining the Fawkes CoT Router Services. This release was officially tagged: "Copper"
1.1.7	Version 1.1.7 includes a number of enhancements. First and foremost services that provide for the management of streaming media have been introduced or enhanced. • The StreamDiscoveryService design and implementation has been completed which allows for the characterization and query capability associated with active streams. The design is fully documented using UML diagrams and a full complement of JUnit tests have been developed to insure that the implementation meets all specifications and design goals. • The Asynchronous ConnectionService design and implementation has been completed. The main function of the connection service is to manage and route streamed data from source to sink. In essence the service acts as a dissemination service for pre-brokered data. A complete set of JUnit tests are included to insure that the implementation meets all specifications and design goals. • The StreamBrokeringService design and implementation has been completed. A complete set of JUnit tests are included to insure that the implementation meets all specifications and design goals. • In addition to the included or enhanced services mentioned above, this release includes the Asynchronous Frame Channels and byte connection groups have been added to the ConnectionService. Current Frame channels are for synchronous designs only. Asynchronous frame channels required for use in the next version of connection service and stream brokering service.

	In addition other features included in Version 1.1.7 include • Long term archival of information instances. This capability allows for the movement of subsets of persisted information instances for RepositoryServices to longer term archive. • Output Channel Monitoring. This implementation, while implemented as a generic solution, is currently applied only to the DisseminationServices. • Input Channel Monitoring. This implementation, while implemented as a generic solution, is currently applied only to the SubmissionServices. • Subscription Monitoring. This implementation utilizes the EventNotificationService and a new Heart-Beat event. • Many other minor bug fixes and enhancements have been made throughout the implementation. As with all code changes, JUnit tests have been created or enhanced to provide adequate This release was officially tagged: "Bronze"
1.1.8	Version 1.1.8 includes a number of enhancements focusing on the introduction of the implementation level Channel Management, Service Multiplexor, and Task Scheduler interfaces and default implementations. • The Channel Manager interface provides a plug-in point for unique channel & buffer management schemes. This capability is exposed at the developer and the Spring configuration levels. • The Service Multiplexor interface provides a plug-in point for policy (or other technology) based logic for determining how services are orchestrated. It is used to determine where an entity (information, event, etc.) goes next after being processed by the parent service. • The Task Scheduler interface provides a plug-in point for prioritizing service task execution. In addition other features included in Version 1.1.8 include • Many other minor bug fixes and enhancements have been made throughout the implementation. As with all code changes, JUnit tests have been created or enhanced to provide adequate This release was officially tagged: "Silver"
1.1.9	Version 1.1.9 includes a number of enhancements focusing on the introduction of the implementation level Control Channel Manager interface and default implementation, the basic implementation of the Filter Management Service, Subscription Service, and Service Brokering Service, and numerous bug fixes within the channel and streaming packages. • The Control Channel Manager interface provides a plug-in point for unique service stub management schemes. This capability is exposed at the developer and the Spring configuration levels.

	This release was officially tagged: "Gold"
1.2.0	Version 1.2.0 includes the final revision of the channel interfaces and implementation, the integration of the Stream Repository Service (XRS) into the baseline, functionality upgrades to the Java Service Container, and the first implementation of the Client Runtime Service (CRS) and Information Discovery Service (IDS). • The channel interfaces were re-designed to provide consistent parent interfaces for specific types of channels. The implementation was re-implemented to better define the set of application and transport protocols resulting in a (hopefully) less confusing implementation design. • The Stream Repository Service was designed and implemented to store streaming information. • The Java Service Container was upgraded to include the notion of service dependencies at start-up time. It was also extended to provide the ability to start each service as its own thread of execution within the JVM. This is an ability that may be toggled on or off via the Spring configuration file. • The Client Runtime Service was implemented as an edge actor proxy service. It abstracts most of the detailed Phoenix constructs and coding away from the edge actor using it. • The Information Discovery Service was implemented to provide a central search mechanism for information type definitions and for finding services that are currently supporting specific types of information. This release was officially tagged: "Platinum"
1.2.1	Version 1.2.1 includes boosted JUnit coverage for all modules, the shifting of the streaming services to their own satellite project, and numerous bug fixes to channels and other components. • The streams services were re-located to their own satellite project to simplfy the Base Implementation codebase. This release was officially tagged: "Platinum Plus"